O'REILLY®



图灵程序设计丛书

第2版

Python网络爬虫权威指南

Web Scraping with Python, 2E

全面介绍网页抓取技术，解决Web数据采集、转换和使用中的诸多常见问题和痛点

[美] 瑞安·米切尔 著
神烦小宝 译



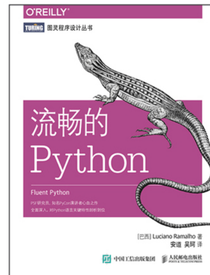
中国工信出版集团



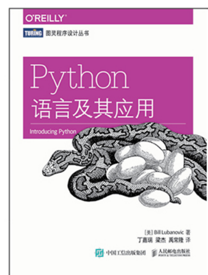
人民邮电出版社
POSTS & TELECOM PRESS

推荐阅读

流畅的Python



Python语言及其应用



Python数据处理



数字版权声明



图灵程序设计丛书

Python网络爬虫权威指南（第2版）

Web Scraping with Python, 2E

[美] 瑞安·米切尔 著
神烦小宝 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（CIP）数据

Python网络爬虫权威指南 / (美) 瑞安·米切尔
(Ryan Mitchell) 著 ; 神烦小宝译. -- 2版. -- 北京 :
人民邮电出版社, 2019.4
（图灵程序设计丛书）
ISBN 978-7-115-50926-0

Ⅰ. ①P… Ⅱ. ①瑞… ②神… Ⅲ. ①软件工具—程序
设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2019)第041375号

内 容 提 要

本书采用简洁强大的 Python 语言，介绍了网页抓取相关技术，并为抓取新式网络中的各种数据类型提供了全面的指导。第一部分重点介绍网页抓取的基本原理：如何用 Python 从网络服务器请求信息，如何对服务器的响应进行基本处理，以及如何以自动化手段与网站进行交互。第二部分介绍如何用网络爬虫测试网站，自动化处理，以及如何通过更多的方式接入网络。

本书适合需要抓取 Web 数据的相关软件开发人员和研究人员阅读。

◆ 著　　[美] 瑞安·米切尔
译　　神烦小宝
责任编辑　岳新欣
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：16.25
字数：384千字　　　　2019年 4 月第 2 版
印数：34 301－38 300册　　　　2019年 4 月北京第 1 次印刷
著作权合同登记号　图字：01-2018-7366号

定价：79.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务还是面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言 xi

第一部分　创建爬虫

第 1 章　初见网络爬虫 3

1.1　网络连接 3

1.2　BeautifulSoup 简介 5

1.2.1　安装 BeautifulSoup 6

1.2.2　运行 BeautifulSoup 8

1.2.3　可靠的网络连接以及异常的处理 9

第 2 章　复杂 HTML 解析 13

2.1　不是一直都要用锤子 13

2.2　再端一碗 BeautifulSoup 14

2.2.1　BeautifulSoup 的 `find()` 和 `find_all()` 16

2.2.2　其他 BeautifulSoup 对象 18

2.2.3　导航树 18

2.3　正则表达式 22

2.4　正则表达式和 BeautifulSoup 25

2.5　获取属性 26

2.6　Lambda 表达式 26

第 3 章　编写网络爬虫 28

3.1　遍历单个域名 28

3.2 抓取整个网站 32
3.3 在互联网上抓取 36

第 4 章 网络爬虫模型 41

4.1 规划和定义对象 41
4.2 处理不同的网站布局 45
4.3 结构化爬虫 49
4.3.1 通过搜索抓取网站 49
4.3.2 通过链接抓取网站 52
4.3.3 抓取多种类型的页面 54
4.4 关于网络爬虫模型的思考 55

第 5 章 Scrapy 57

5.1 安装 Scrapy 57
5.2 创建一个简易爬虫 59
5.3 带规则的抓取 60
5.4 创建 item 64
5.5 输出 item 66
5.6 item 管线组件 66
5.7 Scrapy 日志管理 69
5.8 更多资源 70

第 6 章 存储数据 71

6.1 媒体文件 71
6.2 把数据存储到 CSV 74
6.3 MySQL 75
6.3.1 安装 MySQL 76
6.3.2 基本命令 78
6.3.3 与 Python 整合 81
6.3.4 数据库技术与最佳实践 84
6.3.5 MySQL 里的“六度空间游戏” 86
6.4 Email 88

第二部分 高级网页抓取

第 7 章 读取文档 93

7.1 文档编码 93
7.2 纯文本 94
7.3 CSV 98

7.4 PDF 100
7.5 微软 Word 和 .docx 102

第 8 章 数据清洗 106

8.1 编写代码清洗数据 106
8.2 数据存储后再清洗 111

第 9 章 自然语言处理 115

9.1 概括数据 116
9.2 马尔可夫模型 119
9.3 自然语言工具包 124
9.3.1 安装与设置 125
9.3.2 用 NLTK 做统计分析 126
9.3.3 用 NLTK 做词性分析 128
9.4 其他资源 131

第 10 章 穿越网页表单与登录窗口进行抓取 132

10.1 Python Requests 库 132
10.2 提交一个基本表单 133
10.3 单选按钮、复选框和其他输入 134
10.4 提交文件和图像 136
10.5 处理登录和 cookie 136
10.6 其他表单问题 139

第 11 章 抓取 JavaScript 140

11.1 JavaScript 简介 140
11.2 Ajax 和动态 HTML 143
11.2.1 在 Python 中用 Selenium 执行 JavaScript 144
11.2.2 Selenium 的其他 webdriver 149
11.3 处理重定向 150
11.4 关于 JavaScript 的最后提醒 151

第 12 章 利用 API 抓取数据 152

12.1 API 概述 152
12.1.1 HTTP 方法和 API 154
12.1.2 更多关于 API 响应的介绍 155
12.2 解析 JSON 数据 156
12.3 无文档的 API 157
12.3.1 查找无文档的 API 159
12.3.2 记录未被记录的 API 160
12.3.3 自动查找和记录 API 160

12.4 API 与其他数据源结合 ··· 163
12.5 再说一点 API ··· 165

第 13 章 图像识别与文字处理 ··· 167

13.1 OCR 库概述 ··· 168
13.1.1 Pillow ··· 168
13.1.2 Tesseract ··· 168
13.1.3 NumPy ··· 170
13.2 处理格式规范的文字 ··· 171
13.2.1 自动调整图像 ··· 173
13.2.2 从网站图片中抓取文字 ··· 176
13.3 读取验证码与训练 Tesseract ··· 178
13.4 获取验证码并提交答案 ··· 183

第 14 章 避开抓取陷阱 ··· 186

14.1 道德规范 ··· 186
14.2 让网络机器人看着像人类用户 ··· 187
14.2.1 修改请求头 ··· 187
14.2.2 用 JavaScript 处理 cookie ··· 189
14.2.3 时间就是一切 ··· 191
14.3 常见表单安全措施 ··· 191
14.3.1 隐含输入字段值 ··· 192
14.3.2 避免蜜罐 ··· 192
14.4 问题检查表 ··· 194

第 15 章 用爬虫测试网站 ··· 196

15.1 测试简介 ··· 196
15.2 Python 单元测试 ··· 197
15.3 Selenium 单元测试 ··· 201
15.4 单元测试与 Selenium 单元测试的选择 ··· 205

第 16 章 并行网页抓取 ··· 206

16.1 进程与线程 ··· 206
16.2 多线程抓取 ··· 207
16.2.1 竞争条件与队列 ··· 209
16.2.2 `threading` 模块 ··· 212
16.3 多进程抓取 ··· 214
16.3.1 多进程抓取 ··· 216
16.3.2 进程间通信 ··· 217
16.4 多进程抓取的另一种方法 ··· 219

第 17 章　远程抓取 221
17.1　为什么要用远程服务器 221
17.1.1　避免 IP 地址被封杀 221
17.1.2　移植性与扩展性 222
17.2　Tor 代理服务器 223
17.3　远程主机 224
17.3.1　从网站主机运行 225
17.3.2　从云主机运行 225
17.4　其他资源 227
第 18 章　网页抓取的法律与道德约束 228
18.1　商标、版权、专利 228
18.2　侵害动产 230
18.3　计算机欺诈与滥用法 232
18.4　robots.txt 和服务协议 233
18.5　3 个网络爬虫 236
18.5.1　eBay 起诉 Bidder's Edge 侵害其动产 236
18.5.2　美国政府起诉 Auernheimer 与《计算机欺诈与滥用法》 237
18.5.3　Field 起诉 Google：版权和 robots.txt 239
18.6　勇往直前 239
关于作者 241
关于封面 241

前言

对那些没有学过编程的人来说，计算机编程看着就像变魔术。如果编程是魔术（magic），那么**网页抓取**（Web scraping）就是巫术（wizardry），也就是运用“魔术”来实现精彩实用却又不费吹灰之力的“壮举”。

在我的软件工程师职业生涯中，我几乎没有发现像网页抓取这样的编程实践，可以同时吸引程序员和门外汉的注意。虽然写一个简单的网络爬虫并不难，就是先收集数据，再显示到命令行或者存储到数据库里，但是无论你之前已经做过多少次了，这件事永远会让你感到兴奋，同时又有新的可能。

不过遗憾的是，当和别的程序员提起网页抓取时，我听到了很多关于这件事的误解与困惑。有些人不确定它是不是合法的（其实合法），有些人不明白怎么处理包含大量 JavaScript 的页面以及如何处理登录问题。很多人困惑于如何开始一个大的网页抓取项目，甚至是到哪里寻找他们需要的数据。本书致力于解决人们关于网页抓取的诸多常见问题，廓清一些误解，并对常见的网页抓取任务提供全面的指导。

网页抓取是一个复杂多变的领域，我会通过介绍高级概念以及详细的示例来尽可能地覆盖你可能会在数据抓取项目中遇到的情形。本书提供了代码示例来演示书中的概念，你可以尝试运行它们来实践。这些代码示例是开源的，无论注明出处与否都可以免费使用（但若注明，作者会感激不尽）。所有的代码示例都在 GitHub 网站上（https://github.com/REMitchell/python-scraping），可以查看和下载。

什么是网页抓取

在互联网上进行自动数据抓取这件事和互联网存在的时间差不多一样长。虽然**网页抓取**并不是新术语，但是多年以来，这件事更常见的称谓是**网页抓屏**（screen scraping）、**数据挖掘**（data mining）、**网页收割**（Web harvesting）或其他类似的版本。今天大众好像更倾向

于用“网页抓取”，因此我在本书中使用这个术语，不过我倾向于把遍历多个页面的程序称作**网络爬虫**（Web crawler），或者把网页抓取程序称为**网络机器人**（bot）。

理论上，网页抓取是一种通过多种手段收集网络数据的方式，不光是通过与 API 交互（或者直接与浏览器交互）的方式。最常用的方法是写一个自动化程序向网络服务器请求数据（通常是用 HTML 表单或其他网页文件），然后对数据进行解析，提取需要的信息。

实践中，网页抓取涉及非常广泛的编程技术和手段，比如数据分析、自然语言解析和信息安全等。本书将在第一部分介绍关于网页抓取和网页爬取（crawling）的基础知识，一些高级主题放在第二部分介绍。我建议所有读者仔细学习第一部分，并根据自己的实际需求深入探索第二部分。

为什么要做网页抓取

如果你上网的唯一方式就是用浏览器，那么你其实错过了很多种可能。虽然浏览器可以更方便地执行 JavaScript、显示图片，并且可以以更适合人类阅读的形式展示数据，但是网络爬虫收集和处理大量数据的能力更为卓越。不像狭窄的显示器窗口一次只能让你看一个网页，网络爬虫可以让你一次查看几千甚至几百万个网页。

另外，网络爬虫可以完成传统搜索引擎不能做的事情。用 Google 搜索“飞往波士顿最便宜的航班”，看到的是大量的广告和主流的航班搜索网站。Google 只知道这些网站的网页会显示什么内容，并不知道在航班搜索应用中输入的各种查询的准确结果。但是，设计较好的网络爬虫可以通过抓取大量的网站数据，绘制出飞往波士顿的航班价格随时间变化的图表，告诉你买机票的最佳时间。

你可能会问：“数据不是可以通过 API 获取吗？”（如果你不熟悉 API，请阅读第 12 章。）确实，如果你能找到一个可以解决问题的 API，那会非常给力。它可以非常方便地从一个计算机程序向另一个计算机程序提供格式完好的数据。对于很多类型的数据都可以找到一个 API，比如推文或者维基百科页面。通常，如果有 API 可用，用 API 来获取数据确实比写一个网络爬虫程序更加方便。但是，很多时候你需要的 API 并不存在或者不适用于你的需求，这是因为：

- 你要收集的数据来自不同的网站，没有一个综合多个网站数据的 API；
- 你想要的数据非常小众或不常见，网站不会为你单独创建一个 API；
- 网站没有基础设施或技术能力去创建 API；
- 数据很宝贵 / 被保护起来，不希望广泛传播。

即使 API 已经存在，可能还会有请求内容和次数的限制，API 能够提供的数据类型或者数据格式可能也无法满足你的需求。

这时网页抓取就派上用场了。你在浏览器上看到的内容，大部分都可以通过编写 Python 程序来获取。如果你可以通过程序获取数据，那么就可以把数据存储到数据库里。如果你可以把数据存储到数据库里，自然也就可以将这些数据可视化。

显然，大量的应用场景都会需要这种几乎可以毫无阻碍地获取数据的手段：市场预测、机器语言翻译，甚至医疗诊断领域，通过对新闻网站、文章以及健康论坛中的数据进行抓取和分析，也可以获得很多好处。

甚至在艺术领域，网页抓取也为艺术创作开辟了新方向。由 Jonathan Harris 和 Sep Kamvar 在 2006 年发起的"我们感觉挺好"（We Feel Fine）项目，从大量英文博客中抓取以"I feel"和"I am feeling"开头的短句，最终做成了一个很受大众欢迎的数据可视图，描述了这个世界每天、每分钟的感觉。

无论你现在处于哪个领域，网页抓取都可以让你的工作更高效，帮你提升生产力，甚至开创一个全新的领域。

关于本书

本书不仅介绍了网页抓取，也为抓取、转换和使用新式网络中各种类型的数据提供了全面的指导。虽然本书用的是 Python 编程语言，涉及 Python 的许多基础知识，但这并不是一本 Python 入门书。

如果你完全不了解 Python，那么这本书看起来可能有点儿费劲。请不要将本书用作 Python 的入门书。我尽量按照初、中级 Python 编程水平来编写书中的概念和代码示例，以便让更广泛的读者可以轻松地理解本书。但书中偶尔会包含一些更高级的 Python 编程知识以及一些常见的计算机科学话题。如果你是一位编程高手，那么你可以跳过书中相应的内容。

如果你想更全面地学习 Python，Bill Lubanovic 写的《Python 语言及其应用》[1] 是本非常好的教材，只是书有点儿厚。如果不想看书，Jessica McKellar 的教学视频 Introduction to Python 也非常不错。我也非常喜欢我的前教授 Allen Downey 写的《像计算机科学家一样思考 Python》，这本书非常适合编程新手，介绍了计算机科学和软件工程的概念，以及 Python 语言。

技术书通常仅仅专注于一种语言或者一种技术，但是网页抓取是一个相当分散的主题，在实践中会涉及数据库、网络服务器、HTTP 协议、HTML 语言、网络安全、图像处理、数据科学等内容。本书试图从"数据收集"的角度涵盖所有这些内容以及其他话题。当然，本书不会对这些主题做完整的介绍，但是我相信对于入门编写网络爬虫来说足够了。

第一部分深入讲解网页抓取和网页爬取相关内容，并重点介绍全书都要用到的几个 Python

注 1：中文版已经由人民邮电出版社出版，详见 www.ituring.com.cn/book/1560。——编者注

库。可以将这部分内容用作这些库和技术的综合参考（对于一些特殊情形，后面会提供其他参考资料）。这部分内容对于所有编写网络爬虫的人来说都是实用的，不管网络爬虫的目标或者应用场景如何。

第二部分介绍读者在动手编写网络爬虫的过程中可能会觉得有用的一些主题。不过，这些主题可能并不总是适合所有的爬虫。这些主题的范围特别广泛，无法在一章中道尽玄机。因此，文中提供了许多参考资料来方便读者获取更多的信息。

本书结构清晰，你可直接跳到感兴趣的章节中阅读所需的网页抓取技术。如果一个概念或一段代码在之前的章节中出现过，那么我会明确标注出具体的位置。

排版约定

本书使用了下列排版约定。

- **黑体字**
  表示新术语或重点强调的内容。

- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。

- 斜体等宽字体（*`constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示一般性说明。



该图标表示提示或建议。



该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/REMitchell/python-scraping 下载。

本书是要帮你完成工作的。一般来说，如果书中提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用书中的几个代码片段写一个程序无须获得许可，销售或分发 O’Reilly 图书的示例光盘则需要获得许可；引用书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Web Scraping with Python*, Second Edition by Ryan Mitchell (O’Reilly). Copyright 2018 Ryan Mitchell, 978-1-491-998557-1."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

遗憾的是，纸质书很难保持更新。对于网页抓取来说这更是一个挑战，由于本书用到的很多库、网站以及代码可能偶尔会被修改，所以我们的代码示例可能会运行失败或产生意想不到的结果。如果你需要运行代码示例，请从 GitHub 仓库获取代码并运行，而不是从书中直接复制。我和为本书做贡献的读者（可能也包括你）将尽量及时更新 GitHub 仓库的内容。

除了代码示例，书中还提供了用于演示如何安装和运行软件的终端命令。一般来说，这些命令是适用于 Linux 操作系统的，但是通常也适用于拥有正确配置的 Python 环境并安装了 pip 的 Windows 用户。如果无法运行这些终端命令，我提供了针对所有主流操作系统的命令运行说明，并为 Windows 用户提供了一些外部的参考资料。

O’Reilly Safari



Safari（之前称作 Safari Books Online）是一个针对企业、政府、教育者和个人的会员制培训和参考平台。

会员可以访问来自 250 多家出版商的上千种图书、培训视频、学习路径、互动式教程和精选播放列表，这些出版商包括 O’Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。

要了解更多信息，可以访问 http://www.oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：http://shop.oreilly.com/product/0636920078067.do。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

致谢

和那些基于海量用户反馈诞生的优秀产品一样，如果没有许多协作者、支持者和编辑的帮助，本书可能永远都不会出版。首先要感谢 O’Reilly 团队对这个小众主题图书的大力支持，感谢我的朋友和家人阅读初稿并提出宝贵的建议，还要感谢和我一起在 HedgeServ 奋战的同事们帮我分担了很多工作。

尤其要感谢 Allyson MacDonald、Brian Anderson、Miguel Grinberg 和 Eric VanWyk 的建议、指导和偶尔的爱之深责之切。有一些章节和代码示例是根据他们的建议写成的。

还要感谢 Yale Specht 过去 4 年中在本书两个版本上的无尽耐心，他在最初便鼓励我从事这个项目，并在我的写作过程中对文体提出了宝贵的建议。没有他，这本书可能只用一半时间就能写完，但是不会像现在这么实用。

最后，要感谢 Jim Waldo，是他许多年前给一个小孩邮寄了一个 Linux 机箱和 *The Art and Science of C* 那本书，帮她开启了计算机世界的大门。

电子书

扫描如下二维码，即可购买本书电子版。



第一部分

创建爬虫

本书第一部分重点介绍网页抓取的基本原理：如何用 Python 从网络服务器请求信息，如何对服务器的响应进行基本处理，以及如何以自动化方式与网站交互。最终，你将轻松游弋于网络空间，创建出具有域名切换、信息收集以及信息存储功能的爬虫。

说实话，如果你想以较少的预先投入获取较高的回报，网页抓取肯定是一个值得踏入的神奇领域。大体上，你遇到的 90% 的网页抓取项目使用的都是接下来的 6 章里介绍的技术。这部分内容涵盖了一般人（也包括技术达人）在思考“网络爬虫”时通常的想法：

- 通过网站域名获取 HTML 数据
- 解析数据，获取目标信息
- 存储目标信息
- 如果有必要，移动到另一个网页重复这个过程

这将为你学习本书第二部分中更复杂的项目奠定坚实的基础。不要天真地认为这部分内容没有第二部分里的一些比较高级的项目重要。其实，当你写自己的网络爬虫时，几乎每天都要用到第一部分的所有内容。

第1章

初见网络爬虫

一旦你开始抓取网页，就会感受到浏览器为我们做的所有细节。网页上如果没有 HTML 文本格式层、CSS 样式层、JavaScript 执行层和图像渲染层，乍看起来会有点儿吓人，但是在这一章和下一章中，我们将介绍如何在不借助浏览器帮助的情况下格式化和理解数据。

本章将首先向网络服务器发送 GET 请求（获取网页内容的请求）以获取具体网页，再从网页中读取 HTML 内容，最后做一些简单的信息提取，将我们要寻找的内容分离出来。

1.1 网络连接

如果你没在网络或网络安全上花过太多时间，那么互联网的原理可能看起来有点儿神秘。准确地说，每当打开浏览器连接 http://google.com 的时候，我们不会思考网络正在做什么，而且如今也不必思考。实际上，我认为很神奇的是，计算机接口已经如此先进，让大多数人上网的时候完全不思考网络是如何工作的。

但是，网页抓取需要抛开一些接口的遮挡，不仅是在浏览器层（它如何解释所有的 HTML、CSS 和 JavaScript），有时也包括网络连接层。

我们通过下面的例子让你对浏览器获取信息的过程有一个基本的认识。Alice 有一台网络服务器。Bob 有一台台式机正试图连接到 Alice 的服务器。当一台机器想与另一台机器对话时，下面的某个行为将会发生。

(1) Bob 的电脑发送一串 1 和 0 比特值，表示电路上的高低电压。这些比特构成了一种信息，包括请求头和消息体。请求头包含当前 Bob 的本地路由器 MAC 地址和 Alice 的 IP

地址。消息体包含 Bob 对 Alice 服务器应用的请求。

(2) Bob 的本地路由器收到所有 1 和 0 比特值，把它们理解成一个数据包（packet），从 Bob 自己的 MAC 地址“寄到”Alice 的 IP 地址。他的路由器把数据包“盖上”自己的 IP 地址作为“发件”地址，然后通过互联网发出去。
(3) Bob 的数据包游历了一些中介服务器，沿着正确的物理 / 电路路径前进，到了 Alice 的服务器。
(4) Alice 的服务器在她的 IP 地址收到了数据包。
(5) Alice 的服务器读取数据包请求头里的目标端口，然后把它传递到对应的应用——网络服务器应用。（目标端口通常是网络应用的 80 端口，可以理解成数据包的“房间号”，IP 地址就是“街道地址”）。
(6) 网络服务器应用从服务器处理器收到一串数据，数据是这样的：
   - 这是一个 GET 请求
   - 请求文件 index.html
(7) 网络服务器应用找到对应的 HTML 文件，把它打包成一个新的数据包发送给 Bob，然后通过它的本地路由器发出去，用同样的过程回传到 Bob 的机器上。

瞧！我们就这样实现了互联网。

那么，在这场数据交换中，Web 浏览器是从哪里开始参与的？完全没有参与。其实，在互联网的历史中，浏览器是一个比较新的发明，始于 1990 年的 Nexus 浏览器。

的确，Web 浏览器是一个非常有用的应用，它创建信息的数据包，命令操作系统发送它们，然后把你获取的数据解释成漂亮的图像、声音、视频和文字。但是，Web 浏览器就是代码，而代码可以分解成许多基本组件，可重写、重用，以及做成我们想要的任何东西。Web 浏览器可以让处理器将数据发送到那些对接无线（或有线）网络接口的应用上，但是你可以用短短的 3 行 Python 代码实现这些功能：

```
from urllib.request import urlopen

html = urlopen('http://pythonscraping.com/pages/page1.html')
print(html.read())
```

你可以使用 GitHub 仓库中的 iPython notebook for Chapter 1（https://github.com/REMitchell/python-scraping/blob/master/Chapter01_BeginningToScrape.ipynb）运行以上代码，也可以把上面这段代码保存为 scrapetest.py，然后在终端运行如下命令：

```
$ python scrapetest.py
```

注意，如果你的设备上也安装了 Python 2.x，并且同时运行两个版本的 Python，可能需要直接指明版本才能运行 Python 3.x 代码：

```
$ python3 scrapetest.py
```

这将会输出 http://pythonscraping.com/pages/page1.html 这个网页的全部 HTML 代码。更准确地说，这会输出在域名为 http://pythonscraping.com 的服务器上 < 网络应用根地址 >/pages 文件夹里的 HTML 文件 page1.html 的源代码。

为什么将这些地址理解为“文件”而不是“页面”非常关键呢？现在大多数网页需要加载许多相关的资源文件，可能是图像文件、JavaScript 文件、CSS 文件，或你需要连接的其他各种网页内容。当 Web 浏览器遇到一个标签时，比如 `<img src="cuteKitten.jpg">`，会向服务器发起另一个请求，以获取 cuteKitten.jpg 文件中的数据为用户充分渲染网页。

当然，你的 Python 程序没有返回并向服务器请求多个文件的逻辑，它只能读取你直接请求的单个 HTML 文件。

```
from urllib.request import urlopen
```

上面的代码其实已经表明了它的含义：它查找 Python 的 request 模块（在 `urllib` 库里面），只导入 `urlopen` 函数。

`urllib` 是 Python 的标准库（就是说你不用额外安装就可以运行这个例子），包含了从网页请求数据，处理 cookie，甚至改变像请求头和用户代理这些元数据的函数。我们将在本书中广泛使用 `urllib`，所以建议你读读这个库的 Python 文档。

`urlopen` 用来打开并读取一个从网络获取的远程对象。因为它是一个非常通用的函数（可以轻松读取 HTML 文件、图像文件或其他任何文件流），所以我们将在本书中频繁地使用它。

1.2 BeautifulSoup简介

> “美味的汤，绿色的浓汤，
> 在热气腾腾的盖碗里装！
> 谁不愿意尝一尝，这样的好汤？
> 晚餐用的汤，美味的汤！”

BeautifulSoup 库的名字取自刘易斯·卡罗尔在《爱丽丝梦游仙境》里的同名诗歌。在故事中，这首诗是素甲鱼[1]唱的。

就像它在仙境中的说法一样，BeautifulSoup 尝试化平淡为神奇。它通过定位 HTML 标签来格式化和组织复杂的网页信息，用简单易用的 Python 对象为我们展现 XML 结构信息。

注 1：Mock Turtle，它本身是一个双关语，指英国维多利亚时代的流行菜肴素甲鱼汤，它其实不是用甲鱼而是用牛肉做的，如同中国的豆制品素鸡，名为素鸡，其实与鸡无关。——译者注

1.2.1 安装BeautifulSoup

由于 BeautifulSoup 库不是 Python 标准库，因此需要单独安装。如果你安装过 Python 库，可以使用你最喜爱的安装器并略过本小节，直接阅读 1.2.2 节。

对于还没有安装过 Python 库的新手（或者需要温习的读者）来说，以下介绍的方法将会用于安装本书中的多个库，所以在后面你可能需要回顾本小节。

在本书中，我们将使用 BeautifulSoup 4（也叫 BS4）。Crummy.com 中有 BeautifulSoup 4 的完整安装说明。Linux 系统上的基本安装方法是：

```
$ sudo apt-get install python-bs4
```

对于 macOS 系统，首先用以下命令安装 Python 的包管理器 pip：

```
$ sudo easy_install pip
```

然后运行以下命令来安装库。

```
$ pip install beautifulsoup4
```

另外，注意如果你的设备上同时安装了 Python 2.x 和 Python 3.x，你需要用 `python3` 运行 Python 3.x：

```
$ python3 myScript.py
```

安装包的时候，也要使用这条命令，否则包有可能安装到 Python 2.x 而不是 Python 3.x 里：

```
$ sudo python3 setup.py install
```

如果用 pip 安装，你还可以用 `pip3` 安装 Python 3.x 版本的包：

```
$ pip3 install beautifulsoup4
```

在 Windows 系统上安装包与在 Linux 和 macOS 上安装差不多。从下载页面下载最新的 BeautifulSoup 4 源代码，解压后进入文件，然后执行：

```
> python setup.py install
```

这样就可以了！ BeautifulSoup 将被当作设备上的一个 Python 库。你可以在 Python 终端里导入它测试一下：

```
$ python
> from bs4 import BeautifulSoup
```

如果没有错误，说明导入成功了。

另外，还有一个 Windows 版 pip 的 .exe 格式安装器 ，装了之后你就可以轻松安装和管理包了：

```
> pip install beautifulsoup4
```

用虚拟环境保存库文件

如果你同时负责多个 Python 项目，或者想要轻松打包某个项目及其关联的库文件，再或者你担心已安装的库之间可能有冲突，那么你可以安装一个 Python 虚拟环境来分而治之。

当不用虚拟环境安装一个 Python 库的时候，你实际上是**全局**安装它。这通常需要有管理员权限，或者以 root 身份安装，这个库文件对设备上的每个用户和每个项目来说都是存在的。好在创建虚拟环境非常简单：

```
$ virtualenv scrapingEnv
```

这样就创建了一个叫作 scrapingEnv 的新环境，你需要先激活它再使用：

```
$ cd scrapingEnv/
$ source bin/activate
```

激活环境之后，你会在命令行提示符前面看到环境名称，提醒你当前处于虚拟环境中。后面你安装的任何库和执行的任何程序都在这个环境下运行。

在新建的 scrapingEnv 环境里，可以安装并使用 BeautifulSoup：

```
(scrapingEnv)ryan$ pip install beautifulsoup4
(scrapingEnv)ryan$ python
> from bs4 import BeautifulSoup
>
```

当不再使用虚拟环境中的库时，可以通过 `deactivate` 命令来退出环境：

```
(scrapingEnv)ryan$ deactivate
ryan$ python
> from bs4 import BeautifulSoup
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'bs4'
```

将项目关联的所有库单独放在一个虚拟环境里，还有助于轻松打包整个环境发送给其他人。只要他们机器上安装的 Python 版本和你的相同，你打包的代码就可以直接通过虚拟环境运行，不需要再安装任何库。

尽管本书的例子都不要求你使用虚拟环境，但是请记住，你可以在任何时候激活并使用它。

1.2.2　运行BeautifulSoup

BeautifulSoup 库最常用的对象恰好就是 BeautifulSoup 对象。让我们把本章开头的例子调整一下再运行看看：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page1.html')
bs = BeautifulSoup(html.read(), 'html.parser')
print(bs.h1)
```

输出结果是：

```
<h1>An Interesting Title</h1>
```

这里仅仅返回了页面上的第一个 h1 标签实例。通常情况下，一个页面也只有一个 h1 标签，但是在 Web 中这个惯例经常被打破，因此你应该意识到这里仅仅检索了该标签的第一个实例，而不一定是你寻找的那个。

和前面网页抓取的例子一样，你导入 urlopen 函数，然后调用 html.read() 获取网页的 HTML 内容。除了文本字符串，BeautifulSoup 还可以使用 urlopen 直接返回的文件对象，而不需要先调用 .read() 函数：

```
bs = BeautifulSoup(html, 'html.parser')
```

这样就可以把 HTML 内容传到 BeautifulSoup 对象，转换成下面的结构：

- **html** → <html><head>...</head><body>...</body></html>
  — **head** → <head><title>A Useful Page</title></head>
    — **title** → <title>A Useful Page</title>
  — **body** → <body><h1>An Int...</h1><div>Lorem ip...</div></body>
    — **h1** → <h1>An Interesting Title</h1>
    — **div** → <div>Lorem Ipsum dolor...</div>

可以看出，我们从网页中提取的 <h1> 标签被嵌在 BeautifulSoup 对象结构的第二层（html → body → h1）。但是，当我们从对象里提取 h1 标签的时候，可以直接调用它：

```
bs.h1
```

其实，下面的所有函数调用都可以产生相同的结果：

```
bs.html.body.h1
bs.body.h1
bs.html.h1
```

当你创建一个 BeautifulSoup 对象时，需要传入两个参数：

```
bs = BeautifulSoup(html.read(), 'html.parser')
```

第一个参数是该对象所基于的 HTML 文本，第二个参数指定了你希望 BeautifulSoup 用来创建该对象的解析器。在大多数情况下，你选择任何一个解析器都差别不大。

`html.parser` 是 Python 3 中的一个解析器，不需要单独安装。如果不是特殊场景的需要，本书中都将使用这个解析器。

另一个常用的解析器是 `lxml`，可以通过 pip 命令安装：

```
$ pip3 install lxml
```

BeautifulSoup 使用 `lxml` 解析器时，只需要改变解析器参数：

```
bs = BeautifulSoup(html.read(), 'lxml')
```

和 `html.parser` 相比，`lxml` 的优点在于解析“杂乱”或者包含错误语法的 HTML 代码的性能更优一些。它可以容忍并修正一些问题，例如未闭合的标签、未正确嵌套的标签，以及缺失的头（head）标签或正文（body）标签。`lxml` 也比 `html.parser` 更快，但是考虑到网络本身的速度将总是你最大的瓶颈，在网页抓取中速度并不是一个必备的优势。

`lxml` 的一个缺点是它必须单独安装，并且它依赖于第三方的 C 语言库。相对于 `html.parser` 来说，这可能会导致可移植性和易用性问题。

另外一个常用的 HTML 解析器是 `html5lib`。和 `lxml` 一样，`html5lib` 也是一个具有容错性的解析器，它甚至可以容忍语法更糟糕的 HTML。它也依赖于外部依赖，并且比 `lxml` 和 `html.parser` 都慢。尽管如此，如果你处理的是一些杂乱的或者手写的 HTML 网站，那么该解析器可能是一个不错的选择。

可以通过安装并将 `html5lib` 字符串传递给 `BeautifulSoup` 对象来使用它：

```
bs = BeautifulSoup(html.read(), 'html5lib')
```

希望这个例子可以向你展示 BeautifulSoup 库的强大与简单。其实，任何 HTML（或 XML）文件的任意节点信息都可以被提取出来，只要目标信息的旁边或附近有标签就行。第 2 章将进一步探讨一些更复杂的 BeautifulSoup 函数，还会介绍正则表达式，以及如何把正则表达式用于 BeautifulSoup 以提取网站信息。

1.2.3 可靠的网络连接以及异常的处理

Web 是十分复杂的。网页数据格式不友好、网站服务器死机、目标数据的标签找不到，都是很麻烦的事情。网页抓取最痛苦的遭遇之一，就是爬虫运行的时候你洗洗睡了，梦想着明天一早数据就都会抓取好放在数据库里，结果第二天醒来，你看到的却是一个因某种数

据格式异常导致运行错误的爬虫，在前一天当你不再盯着屏幕去睡觉之后，没过一会儿爬虫就不再运行了。那个时候，你可能想骂发明网站（以及那些奇葩的网络数据格式）的人，但是你真正应该斥责的人是你自己，为什么一开始不估计可能会出现的异常！

让我们看看爬虫 import 语句后面的第一行代码，看看如何处理可能出现的异常：

```
html = urlopen('http://www.pythonscraping.com/pages/page1.html')
```

这行代码主要会发生两种异常：

- 网页在服务器上不存在（或者获取页面的时候出现错误）
- 服务器不存在

发生第一种异常时，程序会返回 HTTP 错误。HTTP 错误可能是“404 Page Not Found”“500 Internal Server Error”等。对于所有类似情形，urlopen 函数都会抛出 HTTPError 异常。我们可以用下面的方式处理这种异常：

```
from urllib.request import urlopen
from urllib.error import HTTPError

try:
    html = urlopen('http://www.pythonscraping.com/pages/page1.html')
except HTTPError as e:
    print(e)
    # 返回空值，中断程序，或者执行另一个方案
else:
    # 程序继续。注意：如果你已经在上面异常捕捉那一段代码里返回或中断（break），
    # 那么就不需要使用else语句了，这段代码也不会执行
```

如果程序返回 HTTP 错误代码，程序就会显示错误内容，不再执行 else 语句后面的代码。

如果服务器不存在（就是说链接 http://www.pythonscraping.com 打不开，或者是 URL 链接写错了），urlopen 会抛出一个 URLError 异常。这就意味着获取不到服务器，并且由于远程服务器负责返回 HTTP 状态代码，所以不能抛出 HTTPError 异常，而且还应该捕获到更严重的 URLError 异常。你可以增加以下检查代码：

```
from urllib.request import urlopen
from urllib.error import HTTPError
from urllib.error import URLError

try:
    html = urlopen('https://pythonscrapingthisurldoesnotexist.com')
except HTTPError as e:
    print(e)
except URLError as e:
    print('The server could not be found!')
else:
    print('It Worked!')
```

当然，即使从服务器成功获取网页，如果网页上的内容并非完全是我们期望的那样，仍然可能会出现异常。每当你调用 BeautifulSoup 对象里的一个标签时，增加一个检查条件以保证标签确实存在是很聪明的做法。如果你想要调用的标签不存在，BeautifulSoup 就会返回 None 对象。不过，如果再调用这个 None 对象下面的子标签，就会发生 AttributeError 错误。

下面这行代码（nonExistentTag 是虚拟的标签，BeautifulSoup 对象里实际没有）

```
print(bs.nonExistentTag)
```

会返回一个 None 对象。处理和检查这个对象是十分必要的。如果你不检查，直接调用这个 None 对象的子标签，就会有麻烦，如下所示。

```
print(bs.nonExistentTag.someTag)
```

这时就会返回一个异常：

```
AttributeError: 'NoneType' object has no attribute 'someTag'
```

那么怎么才能避免这两种情形的异常呢？最简单的方式就是对两种情形进行检查：

```
try:
    badContent = bs.nonExistingTag.anotherTag
except AttributeError as e:
    print('Tag was not found')
else:
    if badContent == None:
        print ('Tag was not found')
    else:
        print(badContent)
```

初看这些检查与错误处理的代码会觉得有点儿累赘，但是我们可以重新简单组织一下代码，让它变得不那么难写（更重要的是，不那么难读）。例如，下面的代码是上面爬虫的另一种写法：

```
from urllib.request import urlopen
from urllib.error import HTTPError
from bs4 import BeautifulSoup

def getTitle(url):
    try:
        html = urlopen(url)
    except HTTPError as e:
        return None
    try:
        bs = BeautifulSoup(html.read(), 'html.parser')
        title = bs.body.h1
    except AttributeError as e:
        return None
```

```
        return title

title = getTitle('http://www.pythonscraping.com/pages/page1.html')
if title == None:
    print('Title could not be found')
else:
    print(title)
```

在这个例子中，我们创建了一个 getTitle 函数，它可以返回网页的标题，如果获取网页的时候遇到问题就返回一个 None 对象。在 getTitle 函数里面，我们像前面那样检查了 HTTPError，还检查了由于 URL 输入错误引起的 URLError，然后把两行 BeautifulSoup 代码封装在一个 try 语句里面。这两行中的任何一行有问题，都可能抛出 AttributeError（如果服务器不存在，html 就是一个 None 对象，html.read() 就会抛出 AttributeError）。其实，我们可以在 try 语句里面放任意多行代码，或者调用一个在任意位置都可以抛出 AttributeError 的函数。

在写爬虫的时候，思考代码的总体格局，让代码既可以捕捉异常又容易阅读，这是很重要的。如果你还希望重用大量代码，那么拥有像 getSiteHTML 和 getTitle 这样的通用函数（具有周密的异常处理功能）会让快速、稳定地抓取网页变得简单易行。

第2章

复杂HTML解析

当米开朗基罗被问及如何完成《大卫》这样匠心独具的雕刻作品时，他有一段著名的回答："很简单，你只要用锤子把石头上不像大卫的地方敲掉就行了。"

虽然网页抓取和大理石雕刻大相径庭，但是当我们从复杂的网页中寻觅信息时，也必须持有类似的态度。有很多技巧可以帮我们"敲掉"网页上那些不需要的信息，直到找到目标信息。这一章将介绍如何解析复杂的 HTML 页面，从中提取出所需的信息。

2.1 不是一直都要用锤子

面对页面解析难题时，很容易不假思索地直接写几行语句来提取信息。但是，像这样鲁莽放纵地使用技术，只会让程序变得难以调试或脆弱不堪，甚至二者兼具。在开始解析网页之前，让我们看一些可以避免解析复杂 HTML 页面的方式。

假如你已经确定了目标内容，可能是一个名字、一组统计数据或者一段文字。你的目标内容可能隐藏在一个 HTML"烂泥堆"的第 20 层标签里，带有许多没用的标签或 HTML 属性。假如你不经考虑地直接写出下面这样一行代码来提取内容：

```
bs.find_all('table')[4].find_all('tr')[2].find('td').find_all('div')[1].find('a')
```

虽然也可以达到目标，但这样看起来并不是很好。除了代码欠缺美感之外，还有一个问题是，即便网站管理员对网站稍作修改，这行代码也会失效，甚至可能会毁掉整个网络爬虫。那么如果网站开发人员决定增加一张表格或者增加一列数据，你应该怎么做呢？如果网站开发人员在页面的顶部增加一个组件（一些 `div` 标签），你应该怎么做呢？以上的代

码是不安全的，它依赖于网站的结构永远不变。

那么你可以怎么做呢？

- 寻找“打印此页”的链接，或者看看网站有没有 HTML 样式更友好的移动版（把自己的请求头设置成处于移动设备的状态，然后接收网站移动版，更多内容在第 14 章介绍）。
- 寻找隐藏在 JavaScript 文件里的信息。要实现这一点，你可能需要查看网页加载的 JavaScript 文件。我曾经在把一个网站上的街道地址（以经度和纬度呈现的）整理成格式整洁的数组时，查看过内嵌谷歌地图的 JavaScript 文件，里面有每个地址的标记点。
- 虽然网页标题经常会用到，但是这个信息也许可以从网页的 URL 链接里获取。
- 如果你要找的信息只存在于一个网站上，别处没有，那你确实是运气不佳。如果不只限于这个网站，那么你可以找找其他数据源。有没有其他网站也显示了同样的数据？网站上显示的数据是不是从其他网站上抓取后攒出来的？

尤其是在面对埋藏很深或格式不友好的数据时，千万不要不经思考就写代码，一定要三思而后行。

如果你确定自己不能另辟蹊径，那么本章其余的内容就是为你准备的。本章接下来会介绍基于位置、上下文、属性和内容选择标签的标准方式和创新方式。这里展示的技巧如果运用得当，将会助你在编写更稳定可靠的网络爬虫的路上走得更远。

2.2 再端一碗BeautifulSoup

在第 1 章里，我们快速演示了 BeautifulSoup 的安装与运行过程，同时也实现了每次选择一个对象的解析方法。这一节将介绍通过属性查找标签的方法，标签组的使用，以及标签解析树的导航过程。

基本上，你遇到的每个网站都有层叠样式表（cascading style sheet，CSS）。虽然你可能会认为，专门为了让浏览器和人类可以理解网站内容而设计一个展现样式的层，是一件愚蠢的事，但是 CSS 的发明却是网络爬虫的福音。CSS 可以让 HTML 元素呈现出差异化，使那些具有完全相同修饰的元素呈现出不同的样式。比如，有些标签看起来是这样：

```
<span class="green"></span>
```

而另一些标签看起来是这样：

```
<span class="red"></span>
```

网络爬虫可以通过 `class` 属性的值，轻松地区分出两种不同的标签。例如，它们可以用 BeautifulSoup 抓取网页上所有的红色文字，而绿色文字一个都不抓。因为 CSS 通过属性准确地呈现网站的样式，所以你大可放心，大多数现代网站上的 `class` 和 `id` 属性资源都非常丰富。

下面让我们创建一个网络爬虫来抓取 http://www.pythonscraping.com/pages/warandpeace.html 这个网页。

在这个页面里，小说人物的对话内容都是红色的，人物名称都是绿色的。你可以看到网页源代码里的 span 标签引用了对应的 CSS 属性，如下所示：

```
<span class="red">Heavens! what a virulent attack!</span> replied
<span class="green">the prince</span>, not in the least disconcerted
by this reception.
```

我们可以使用和第 1 章类似的程序抓取整个页面，然后创建一个 BeautifulSoup 对象：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page1.html')
bs = BeautifulSoup(html.read(), 'html.parser')
```

通过 BeautifulSoup 对象，我们可以用 find_all 函数提取只包含在 标签里的文字，这样就会得到一个人物名称的 Python 列表（find_all 是一个非常灵活的函数，后面会经常用到它）：

```
nameList = bs.findAll('span', {'class':'green'})
for name in nameList:
    print(name.get_text())
```

代码执行以后就会按照《战争与和平》中的人物出场顺序显示所有的人名。这是怎么实现的呢？之前，我们调用 bs.tagName 只能获取页面中指定的第一个标签。现在，调用 bs.find_all(tagName, tagAttributes) 可以获取页面中所有指定的标签，不再只是第一个了。

获取人名列表之后，程序遍历列表中所有的名字，然后打印 name.get_text()，就可以把标签中的内容分开显示了。



什么时候使用 get_text()？什么时候应该保留标签？

.get_text() 会清除你正在处理的 HTML 文档中的所有标签，然后返回一个只包含文字的 Unicode 字符串。假如你正在处理一个包含许多超链接、段落和其他标签的大段文本，那么 .get_text() 会把这些超链接、段落和标签都清除掉，只剩下一串不带标签的文字。

用 BeautifulSoup 对象查找你想要的信息，比直接在 HTML 文本里查找信息要简单得多。通常在你准备打印、存储和操作最终数据时，应该最后才使用 .get_text()。一般情况下，你应该尽可能地保留 HTML 文档的标签结构。

2.2.1 BeautifulSoup的find()和find_all()

BeautifulSoup 里的 find() 和 find_all() 可能是你最常用的两个函数。借助它们，你可以通过标签的不同属性轻松地过滤 HTML 页面，查找需要的标签组或单个标签。

这两个函数非常相似，BeautifulSoup 文档里两者的定义就是这样：

```
find_all(tag, attributes, recursive, text, limit, keywords)
find(tag, attributes, recursive, text, keywords)
```

很可能你会发现，自己在 95% 的时间里都只需要使用前两个参数：tag 和 attributes。但是，我们还是应该仔细地看看所有的参数。

标签参数 tag 前面已经介绍过——你可以传递一个标签的名称或多个标签名称组成的 Python 列表做标签参数。例如，下面的代码将返回一个包含 HTML 文档中所有标题标签的列表：[1]

```
.find_all(['h1','h2','h3','h4','h5','h6'])
```

属性参数 attributes 用一个 Python 字典封装一个标签的若干属性和对应的属性值。例如，下面这个函数会返回 HTML 文档里红色与绿色两种颜色的 span 标签：

```
.find_all('span', {'class':{'green', 'red'}})
```

递归参数 recursive 是一个布尔变量。你想抓取 HTML 文档标签结构里多少层的信息？如果 recursive 设置为 True，find_all 就会根据你的要求去查找标签参数的所有子标签，以及子标签的子标签。如果 recursive 设置为 False，find_all 就只查找文档的一级标签。find_all 默认是支持递归查找的（recursive 默认值是 True）；一般情况下这个参数不需要设置，除非你真正了解自己需要哪些信息，而且抓取速度非常重要，那时你可以设置递归参数。

文本参数 text 有点不同，它是用标签的文本内容去匹配，而不是用标签的属性。假如我们想查找前面网页中包含“the prince”内容的标签数量，可以把之前的 find_all 方法换成下面的代码：

```
nameList = bs.find_all(text='the prince')
print(len(nameList))
```

输出结果为“7”。

范围限制参数 limit 显然只用于 find_all 方法。find 其实等价于 limit 等于 1 时的 find_all。

注 1：如果你想获得文档里的一组 h<some_level> 标签，可以用更简洁的方法写代码来完成。我们将在 2.3 节介绍这类问题的其他处理方法。

如果你想获取网页中的前 *x* 项结果，就可以设置它。但是要注意，设置这个参数之后，获得的前几项结果是按照网页上的顺序排序的，未必是你想要的那前几项。

还有一个关键词参数 keyword，可以让你选择那些具有指定属性的标签。例如：

```
title = bs.find_all(id='title', class_='text')
```

上述代码返回第一个在 class_ 属性中包含单词 text 并且在 id 属性中包含 title 的标签。需要注意的是，通常情况下，页面中每个 id 的属性值只能被使用一次。因此在实际情况中，上面的代码可能并不实用，而以下代码可以达到同样的效果：

```
title = bs.find(id='title')
```

关键词参数和“类”的注意事项

虽然关键词参数 keyword 在一些场景中很有用，但是，它实际上是一个冗余的 BeautifulSoup 功能。任何用关键词参数能够完成的任务，同样可以用本章后面将介绍的技术解决（请参见 2.3 节和 2.6 节）。

例如，下面两行代码是完全一样的：

```
bs.find_all(id='text')
bs.find_all('', {'id':'text'})
```

另外，用 keyword 偶尔会出现问题，尤其是在用 class 属性查找标签的时候，因为 class 是 Python 中受保护的关键字。也就是说，class 是 Python 语言的保留字，在 Python 程序里是不能当作变量或参数名使用的（和前面介绍的 BeautifulSoup.find_all() 里的 keyword 无关）[2]。假如你运行下面的代码，Python 就会因为你误用 class 保留字而产生一个语法错误：

```
bs.find_all(class='green')
```

不过，你可以用 BeautifulSoup 提供的有点儿臃肿的方案，在 class 后面增加一个下划线：

```
bs.find_all(class_='green')
```

另外，你也可以用属性参数把 class 用引号包起来：

```
bs.find_all('', {'class':'green'})
```

看到这里，你可能会扪心自问：“现在我是不是已经知道如何用标签属性获取一组标签了——用字典把属性传到函数里就行了？”

注 2：Python 语言参考里提供了完整的受保护关键字列表。

回忆一下前面的内容，通过标签参数 tag 把标签列表传到 .find_all() 里获取一组标签，其实就是一个“或”关系的过滤器（即选择所有带标签 1、标签 2 或标签 3……的标签）。如果你的标签列表很长，就需要花很长时间才能写完。而关键词参数 keyword 可以让你增加一个“与”关系的过滤器来简化工作。

2.2.2 其他BeautifulSoup对象

看到这里，你已经见过 BeautifulSoup 库里的两种对象了。

BeautifulSoup对象

前面代码示例中的 bs。

标签Tag对象

BeautifulSoup 对象通过 find 和 find_all，或者直接调用子标签获取的一列对象或单个对象，就像：

```
bs.div.h1
```

但是，这个库还有另外两种对象，虽然不常用，却应该了解一下。

NavigableString对象

用来表示标签里的文字，而不是标签本身（有些函数可以操作和生成 NavigableString 对象，而不是标签对象）。

Comment对象

用来查找 HTML 文档的注释标签，<!-- 像这样 -->。

这 4 个对象是你用 BeautifulSoup 库时会遇到的所有对象（写作本书的时候）。

2.2.3 导航树

find_all 函数通过标签的名称和属性来查找标签 。但是如果你需要通过标签在文档中的位置来查找标签，该怎么办？这就是导航树（navigating trees）的作用。在第 1 章里，我们看过用单一方向进行 BeautifulSoup 标签树导航：

```
bs.tag.subTag.anotherSubTag
```

现在我们用虚拟的在线购物网站 http://www.pythonscraping.com/pages/page3.html 作为要抓取的示例网页，演示 HTML 导航树的纵向和横向导航（如图 2-1 所示）。



Totally Normal Gifts

Here is a collection of totally normal, totally reasonable gifts that your friends are sure to love! Our collection is hand-curate

We haven't figured out how to make online shopping carts yet, but you can send us a check to:
123 Main St.
Abuja, Nigeria
We will then send your totally amazing gift, pronto! Please include an extra $5.00 for gift wrapping.

Item Title	Description	Cost	Image
Vegetable Basket	This vegetable basket is the perfect gift for your health conscious (or overweight) friends! ***Now with super-colorful bell peppers!***	$15.00	
Russian Nesting Dolls	Hand-painted by trained monkeys, these exquisite dolls are priceless! And by "priceless," we mean "extremely expensive"! ***8 entire dolls per set! Octuple the presents!***	$10,000.52	
Fish Painting	If something seems fishy about this painting, it's because it's a fish! ***Also hand-painted by trained monkeys!***	$10,005.00	
Dead Parrot	This is an ex-parrot! ***Or maybe he's only resting?***	$0.50	

图 2-1：http://www.pythonscraping.com/pages/page3.html 截图

这个 HTML 页面可以映射成一棵树（为了简洁，省略了一些标签），如下所示。

```
• HTML
  — body
     — div.wrapper
        — h1
        — div.content
        — table#giftList
           — tr
              — th
              — th
              — th
              — th
           — tr.gift#gift1
              — td
              — td
                 — span.excitingNote
              — td
              — td
                 — img
           — ……其他表格行省略了……
        — div.footer
```

在后面几节内容里，我们仍然以这个 HTML 标签结构为例。

1. 处理子标签和其他后代标签

在计算机科学和一些数学领域中，你经常会听到“虐子”事件（比喻对一些子事件的处理方式）：移动它们，储存它们，删除它们，甚至杀死它们。值得庆幸的是，这里只选择它们。

和许多其他库一样，在 BeautifulSoup 库里，**孩子**（child）和**后代**（descendant）有显著的不同：和人类的家谱一样，子标签就是父标签的下一级，而后代标签是指父标签下面所有级别的标签。例如，`tr` 标签是 `table` 标签的子标签，而 `tr`、`th`、`td`、`img` 和 `span` 标签都是 `table` 标签的后代标签（我们的示例页面中就是如此）。所有的子标签都是后代标签，但不是所有的后代标签都是子标签。

一般情况下，BeautifulSoup 函数总是处理当前标签的后代标签。例如，`bs.body.h1` 选择了 `body` 标签后代里的第一个 `h1` 标签，不会去找 `body` 外面的标签。

类似地，`bs.div.find_all("img")` 会找出文档中的第一个 `div` 标签，然后获取这个 `div` 后代里所有 `img` 标签的列表。

如果你只想找出子标签，可以用 `.children` 标签：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')

for child in bs.find('table',{'id':'giftList'}).children:
    print(child)
```

这段代码会打印 `giftList` 表格中所有产品的数据行，包括最开始的列名行。如果你用 `descendants()` 函数而不是 `children()` 函数，那么就会打印出二十几个标签，包括 `img` 标签、`span` 标签，以及每个 `td` 标签。掌握子标签与后代标签的差别十分重要！

2. 处理兄弟标签

BeautifulSoup 的 `next_siblings()` 函数使得从表格中收集数据非常简单，尤其是带标题行的表格：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')

for sibling in bs.find('table', {'id':'giftList'}).tr.next_siblings:
    print(sibling)
```

这段代码会打印产品表格里所有行的产品，第一行表格标题除外。为什么标题行被跳过了呢？对象不能是自己的兄弟标签。任何时候你获取一个标签的兄弟标签，都不会包含这个标签本身。正如函数名本身揭示的，这个函数只调用**后面的**兄弟标签。例如，如果我们选择一组标签中位于中间位置的一个标签，然后调用 `next_siblings()` 函数，那么就只会返回在它后面的兄弟标签。因此，选择标题行，然后调用 `next_siblings`，就可以选择表格中除了标题行以外的所有行。



让标签的选择更具体

如果我们选择 `bs.table.tr` 或直接用 `bs.tr` 来获取表格中的第一行，上面的代码也可以获得正确的结果。但是，我还是写了一行更长、更完整的代码：

```
bs.find('table',{'id':'giftList'}).tr
```

即使页面上只有一个表格（或其他目标标签），只用标签也很容易丢失细节。另外，页面布局是不断变化的。一个标签这次是在表格中第一行的位置，没准儿哪天就在第二行或第三行了。如果想让你的爬虫更稳定，最好还是让标签的选择更加具体。如果有属性，就利用标签的属性。

和 `next_siblings` 一样，如果你很容易找到一组兄弟标签中的最后一个标签，那么 `previous_siblings` 函数也会很有用。

当然，还有 `next_sibling` 和 `previous_sibling` 函数，它们的作用跟 `next_siblings` 和 `previous_siblings` 类似，只是它们返回的是单个标签，而不是一组标签。

3. 处理父标签

在抓取网页的时候，查找父标签的需求比查找子标签和兄弟标签要少很多。通常情况下，如果以抓取网页内容为目的来观察 HTML 页面，我们都是从最上层标签开始的，然后思考如何定位我们想要的数据块所在的位置。但是，偶尔在特殊情况下你也会用到 BeautifulSoup 的父标签查找函数 `parent` 和 `parents`。例如：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')
print(bs.find('img',
              {'src':'../img/gifts/img1.jpg'})
      .parent.previous_sibling.get_text())
```

这段代码会打印 ../img/gifts/img1.jpg 这个图片所对应商品的价格（这个示例中价格是 $15.00）。

这是如何实现的呢？下面是我们正在处理的 HTML 页面的部分结构，其中用数字表示了步骤。

- <tr>
  — td
  — td
  — td ❸
    — "$15.00" ❹
  — td ❷
    — <img src="../img/gifts/img1.jpg"> ❶

❶ 首先选择图片标签 src="../img/gifts/img1.jpg"。

❷ 选择图片标签的父标签（在示例中是 td 标签）。

❸ 选择 td 标签的前一个兄弟标签 previous_sibling（在示例中是包含美元价格的 td 标签）。

❹ 选择标签中的文字，“$15.00”。

2.3　正则表达式

计算机科学领域有个笑话：“如果你有一个问题打算用正则表达式来解决，那么就是两个问题了。”

不幸的是，正则表达式（通常简写 regex）经常被嘲笑是一堆随机符号混杂在一起，看起来毫无意义。这种印象让人对其避而远之，然后费尽心思写一堆复杂的查找和过滤函数，其实他们真正需要的就是一行正则表达式。

其实正则表达式上手一点儿也不难，而且运行很快，通过一些简单的例子就可以轻松地学会。

之所以叫**正则表达式**，是因为它们可以识别正则字符串（regular string）；也就是说，它们可以这么定义：“如果你给我的字符串符合规则，我就返回它”，或者是“如果字符串不符合规则，我就忽略它”。这在快速浏览大文档，以查找像电话号码和邮箱地址之类的字符串时，是非常方便的。

注意这里我用了一个词组**正则字符串**。什么是正则字符串？其实就是任意可以用一系列线性规则构成的字符串[3]，就像：

(1) 字母“a”至少出现一次；
(2) 后面跟着字母“b”，重复 5 次；
(3) 后面再跟字母“c”，重复任意偶数次；

注 3：你可能会问：“有没有‘非正则’的表达式？”非正则表达式超出了本书的介绍范围，它们其实是指那些像“前面是素数个 a，后面跟着两倍于 a 数量的 b”“写一个回文”之类的字符串。用正则表达式不可能写出这类字符串。不过好在我的网络爬虫至今还从未遇到过这种需求。

(4) 最后一位是字母“d”或“e”。

满足上面规则的字符串有“aaaabbbbbccccd”“aabbbbbcce”等（有无穷多种变化）。

正则表达式就是表达这组规则的一种快捷方式。这组规则的正则表达式如下所示：

```
aa*bbbbb(cc)*(d|e)
```

乍看这个字符串会觉得有点儿奇葩，但是当我们把它分解之后就会很清楚了。

`aa*`
: a 后面跟着的 a*（读作 a 星）表示“重复任意次 a，包括 0 次”。这样就可以保证字母 a 至少出现一次。

`bbbbb`
: 这没什么特别的，就是 5 个 b。

`(cc)*`
: 任意偶数个字符都可以编组，这个规则是用括号括住两个 c，然后后面跟一个星号，表示可以有任意次两个 c（也可以是 0 次）。

`(d|e)`
: 在两个表达式中间增加一个竖线（|）表示“这个或那个”。本例中表示“增加一个 d 或者一个 e”。这样就可以保证字符串的结尾是这两个字母之一。



尝试正则表达式

在学习书写正则表达式的时候，通过实验来感受一下它们如何工作，这是至关重要的。如果你不想打开代码编辑器，写几行代码，然后再运行程序以检查正则表达式的运行是否符合预期，那么你可以去 Regex Pal 这类网站在线测试你的正则表达式。

表 2-1 列出了常用的正则表达式符号，以及简短的解释和示例。这个列表并没有囊括全部的正则表达式，正如前面提到的，不同语言中的正则表达式符号会略有不同。但是，这里列出的 12 个符号是 Python 中最常用的正则表达式符号，可以用于查找和获取几乎任意字符串类型。

表2-1：常用的正则表达式符号

符号	含　义	例　子	匹配结果
*	匹配前面的字符、子表达式或括号里的字符 0 次或多次	a*b*	aaaaaaaa，aaabbbbb，bbbbbb
+	匹配前面的字符、子表达式或括号里的字符至少 1 次	a+b+	aaaaaaab，aaabbbbb，abbbbbb

（续）

<table>
<tr><th>符号</th><th>含　　义</th><th>例　　子</th><th>匹配结果</th></tr>
<tr><td>[]</td><td>匹配中括号里的任意一个字符（相当于“任选一个”）</td><td>[A-Z]*</td><td>APPLE，CAPITALS，QWERTY</td></tr>
<tr><td>()</td><td>表达式编组（在正则表达式的规则里编组会优先运行）</td><td>(a*b)*</td><td>aaabaab，abaaab，ababaaaaab</td></tr>
<tr><td>{m,n}</td><td>匹配前面的字符、子表达式或括号里的字符 m 到 n 次（包含 m 或 n）</td><td>a{2,3}b{2,3}</td><td>aabbb，aaabbb，aabb</td></tr>
<tr><td>[^]</td><td>匹配任意一个不在中括号里的字符</td><td>[^A-Z]*</td><td>apple，lowercase，qwerty</td></tr>
<tr><td>|</td><td>匹配任意一个由竖线分割的字符、子表达式（注意是竖线，不是大字字母 I）</td><td>b(a|i|e)d</td><td>bad，bid，bed</td></tr>
<tr><td>.</td><td>匹配任意单个字符（包括符号、数字和空格等）</td><td>b.d</td><td>bad，bzd，b$d，b d</td></tr>
<tr><td>^</td><td>指字符串开始位置的字符或子表达式</td><td>^a</td><td>apple，asdf，a</td></tr>
<tr><td>\</td><td>转义字符（把有特殊含义的字符转换成字面形式）</td><td>\.\|\\</td><td>.|\</td></tr>
<tr><td>$</td><td>经常用在正则表达式的末尾，表示“从字符串的末端匹配”。如果不用它，每个正则表达式实际都带着“.*”模式，只会从字符串开头进行匹配。这个符号可以看成是 ^ 符号的反义词</td><td>[A-Z]*[a-z]*$</td><td>ABCabc，zzzyx，Bob</td></tr>
<tr><td>?!</td><td>“不包含”。这个奇怪的组合通常放在字符或正则表达式前面，表示字符不能出现在目标字符串里。这个符号比较难用，毕竟字符通常会在字符串的不同部位出现。如果要在整个字符串中彻底排除某个字符，就加上 ^ 和 $ 符号</td><td>^((?![A-Z]).)*$</td><td>no-caps-here，$ymb0ls a4e f!ne</td></tr>
</table>

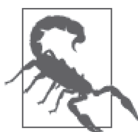
正则表达式在实际中的一个经典应用是识别邮箱地址。虽然不同邮箱服务器的邮箱地址的具体规则不尽相同，但是我们还是可以创建几条通用规则。每条规则对应的正则表达式如下表第 2 列所示。

<table>
<tr><th>规　　则</th><th>正则表达式</th></tr>
<tr><td>1. 邮箱地址的第一部分至少包括一种内容：大写字母、小写字母、数字 0-9、点号（.）、加号（+）或下划线（_）</td><td>[A-Za-z0-9\._+]+：这个正则表达式简写非常智慧。例如，它用“A-Z”表示“A-Z 中的任意大写字母”。把所有可能的序列和符号放在中括号（不是小括号）里表示“可以是方括号中的任何一个符号”。要注意后面的加号，它表示“这些符号都可以出现多次，但至少要出现 1 次”</td></tr>
<tr><td>2. 之后，邮箱地址会包含一个 @ 符号</td><td>@：这个符号很简单：@ 符号必须出现在中间位置，并且只能出现 1 次</td></tr>
<tr><td>3. 在符合 @ 之后，邮箱地址还必须至少包含一个大写或小写字母</td><td>[A-Za-z]+：可能只在域名的前半部分、符号 @ 后面用字母。而且，至少要有一个字符</td></tr>
<tr><td>4. 之后跟一个点号（.）</td><td>\.：在域名前必须有一个点号（.）。退格在这里用作转义字符</td></tr>
<tr><td>5. 最后邮箱地址用 com、org、edu、net 结尾（实际上，顶级域名有很多种可能，但是作为示例演示这 4 个后缀够用了）</td><td>(com|org|edu|net)：这样列出了后半部分邮箱地址中可能出现在点号之后的字母序列</td></tr>
</table>

把上面的规则连接起来，就获得了完整的正则表达式：

```
[A-Za-z0-9\._+]+@[A-Za-z]+\.(com|org|edu|net)
```

当动手开始写正则表达式的时候，最好先写一个步骤列表，具体描述出你的目标字符串结构。还要注意一些细节的处理。比如，当你识别电话号码的时候，会考虑国家代码和分机号吗？



正则表达式：并非处处正则！

正则表达式的标准版（本书使用的版本，用于 Python 和 BeautifulSoup）是基于 Perl 语法演变而来的。绝大多数现代编程语言都使用与之相同或近似的版本。但是要注意，在其他语言中使用这些正则表达式时可能会出问题。有些语言，比如 Java，其正则表达式就和 Python 不太一样。总之，遇到问题时看文档！

2.4　正则表达式和BeautifulSoup

如果你觉得前面介绍的正则表达式内容与本书的主题有点儿脱节，那么这里就把它们连接起来。在抓取网页的时候，BeautifulSoup 和正则表达式总是配合使用的。其实，大多数支持字符串参数的函数（比如，`find(id="aTagIdHere")`）也都支持正则表达式。

让我们看几个例子，待抓取的网页是 http://www.pythonscraping.com/pages/page3.html。

注意观察网页上有几张商品图片，它们的源代码形式如下：

```
<img src="../img/gifts/img3.jpg">
```

如果我们想抓取所有图片的 URL 链接，非常直接的做法就是用 `find_all("img")` 抓取所有图片，对吗？但是有个问题。除了那些明显“多余的”图片（比如 LOGO）之外，现代网站里都有一些隐藏的图片、用于网页布局留白和元素对齐的空白图片，以及一些不容易察觉到的图片标签。总之，你不能仅用商品图片来统计网页上所有的图片。

网页的布局也可能会变化，或者，因为某些原因，我们不想通过图片在网页中的**位置**来查找标签。当你想抓取随机分布在网站里的某个元素或数据时，可能就是这种情况。例如，一些网页的最上面可能有一张布局特殊的商品图片，但是另一些网页上则没有。

解决这类问题的办法，就是直接定位那些标签来查找信息。在本例中，我们直接通过商品图片的文件路径来查找：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
```

```
html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')
images = bs.find_all('img',
    {'src':re.compile('\.\.\/img\/gifts\/img.*\.jpg')})
for image in images:
    print(image['src'])
```

这段代码会打印出图片的相对路径，都是以 ../img/gifts/img 开头，以 .jpg 结尾，其结果如下所示：

```
../img/gifts/img1.jpg
../img/gifts/img2.jpg
../img/gifts/img3.jpg
../img/gifts/img4.jpg
../img/gifts/img6.jpg
```

正则表达式可以作为 BeautifulSoup 语句的任意一个参数，让你可以灵活地查找目标元素。

2.5　获取属性

到目前为止，我们已经介绍过如何获取和过滤标签，以及如何获取标签里的内容。但是，在抓取网页时你经常不需要查找标签的内容，而是需要查找标签属性。比如标签 `a` 指向的 URL 链接包含在 `href` 属性中，或者 `img` 标签的图片文件包含在 `src` 属性中，这时获取标签属性就变得非常有用了。

对于一个标签对象，可以用下面的代码获取它的全部属性：

```
myTag.attrs
```

要注意这行代码返回的是一个 Python 字典对象，可以轻松获取和操作这些属性。比如要获取图片的源位置 `src`，可以用下面这行代码：

```
myImgTag.attrs['src']
```

2.6　Lambda表达式

如果你在学校读的是计算机科学专业，那么可能学过 Lambda 表达式，不过可能从来没有用过它。如果你不是计算机科学专业，它们看着可能有点儿陌生（或者只是"曾经学习过的东西"）。在这一节里，虽然我们不打算深入学习这类函数，但是会用几个例子来演示它们是如何用在网页抓取中的。

Lambda 表达式本质上就是一个函数，可以作为变量传入另一个函数；也就是说，一个函数不是定义成 `f(x, y)`，而是可以定义成 `f(g(x), y)` 或 `f(g(x), h(y))` 的形式。

BeautifulSoup 允许我们把特定类型的函数作为参数传入 `find_all` 函数。唯一的限制条件是这些函数必须把一个标签对象作为参数并且返回布尔类型的结果。BeautifulSoup 用这个函数来评估它遇到的每个标签对象，最后把评估结果为“真”的标签保留，把其他标签剔除。

例如，下面的代码就是获取有两个属性的所有标签：

```
bs.find_all(lambda tag: len(tag.attrs) == 2)
```

这里，作为参数传入的函数是 `len(tag.attrs) == 2`。当该参数为真时，`find_all` 函数将返回 `tag`。即找出带有两个属性的所有标签，如下所示：

```
<div class="body" id="content"></div>
<span style="color:red" class="title"></span>
```

Lambda 函数非常实用，你甚至可以用它来替代现有的 BeautifulSoup 函数：

```
bs.find_all(lambda tag: tag.get_text() ==
    'Or maybe he\'s only resting?')
```

如果不使用 Lambda 函数，代码如下：

```
bs.find_all('', text='Or maybe he\'s only resting?')
```

如果你能记住 Lambda 函数的语法，以及如何获取标签的属性，那么你可能再也不需要记住 BeautifulSoup 的语法了！

由于 Lambda 函数可以是任意返回 `True` 或者 `False` 值的函数，你甚至可以结合使用 Lambda 函数与正则表达式，来查找匹配特定字符串模式的属性的标签。

第3章

编写网络爬虫

到目前为止，本书的例子都只是处理单个静态页面，只能算是人为简化的例子（使用作者的网站页面）。从本章开始，我们会看到一些现实问题，需要用爬虫遍历多个页面甚至多个网站。

之所以叫**网络爬虫**，是因为它们可以在 Web 上爬行。它们本质上就是一种递归方式。它们必须首先获取一个 URL 对应的网页内容，然后检查这个页面，寻找另一个 URL，再获取该 URL 对应的网页内容，并不断循环这一过程。

不过要注意的是：你可以抓取网页，并不意味着你总是应该这么做。当你需要的所有数据都在一个页面上时，前面例子中的爬虫就足以解决问题了。使用网络爬虫的时候，必须非常谨慎地考虑需要消耗多少带宽，还要尽力思考能不能让抓取目标的服务器负载更低一些。

3.1　遍历单个域名

即使你没听说过“维基百科六度分隔理论”，也很可能听过“凯文·贝肯（Kevin Bacon）的六度分隔值游戏”。在这两个游戏中，目标都是把两个不相干的主题（在前一种情况中是相互链接的维基百科词条，而在后一种情况中是出现在同一部电影中的演员）用一个链条（至多包含 6 个主题，包括原来的两个主题）连接起来。

比如，埃里克·艾德尔和布兰登·弗雷泽都出现在电影《骑警杜德雷》里，布兰登·弗雷泽又和凯文·贝肯都出现在电影《我呼吸的空气》里。[1] 因此，根据这两个条件，从埃里克·艾德尔到凯文·贝肯的链条长度只有 3 个主题。

注 1：感谢 The Oracle of Bacon 的存在，满足了我对这类关系链的好奇心。

我们将在本节创建一个项目来实现“维基百科六度分隔理论”的查找方法。也就是说，我们要实现从埃里克·艾德尔的词条页面（https://en.wikipedia.org/wiki/Eric_Idle）开始，经过最少的链接点击次数找到凯文·贝肯的词条页面（https://en.wikipedia.org/wiki/Kevin_Bacon）。

> **这么做对维基百科的服务器负载有多大影响？**
>
> 根据维基媒体基金会（维基百科所属的组织）的统计，该网站每秒会收到大约 2500 次点击，其中超过 99% 的点击都指向维基百科域名［详情请见“维基媒体统计图”（Wikimedia in Figures）里的“流量数据”（Traffic Volume）部分内容］。因为网站流量很大，所以你的网络爬虫不可能对维基百科的服务器负载产生显著影响。不过，如果你频繁地运行本书的代码示例，或者自己创建项目来抓取维基百科的词条，那么希望你能够向维基媒体基金会提供一点捐赠——不只是为了抵消你占用的服务器资源，也是为了其他人能够利用维基百科这个教育资源。
>
> 还需要注意的是，如果你准备利用维基百科的数据做一个大型项目，应该确认该数据是不能够通过维基百科 API 获取的。维基百科网站经常被用于演示爬虫，因为它的 HTML 结构简单并且相对稳定。但是它的 API 往往会使得数据获取更加高效。

你应该已经知道如何写一段 Python 代码，来获取维基百科网站的任何页面并提取该页面中的链接了。

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://en.wikipedia.org/wiki/Kevin_Bacon')
bs = BeautifulSoup(html, 'html.parser')
for link in bs.find_all('a'):
    if 'href' in link.attrs:
        print(link.attrs['href'])
```

如果你观察生成的一列链接，会看到你想要的所有词条链接都在里面：“Apollo 13”“Philadelphia”“Primetime Emmy Award”，等等。但是，也有一些你不需要的链接：

```
//wikimediafoundation.org/wiki/Privacy_policy
//en.wikipedia.org/wiki/Wikipedia:Contact_us
```

其实，维基百科的每个页面都充满了侧边栏、页眉和页脚链接，以及连接到分类页面、对话页面和其他不包含词条的页面的链接：

```
/wiki/Category:Articles_with_unsourced_statements_from_April_2014
/wiki/Talk:Kevin_Bacon
```

最近我有个朋友在做一个类似的维基百科抓取项目，他说，为了判断一个维基百科内链是否链接到一个词条页面，他写了一个很大的过滤函数，代码超过了 100 行。不幸的是，他没有提前花很多时间去寻找“词条链接”和“其他链接”之间的模式，也可能他后来发现

了。如果你仔细观察那些指向词条页面（不是指向其他内部页面）的链接，会发现它们都有 3 个共同点：

- 它们都在 id 是 bodyContent 的 div 标签里
- URL 不包含冒号
- URL 都以 /wiki/ 开头

我们可以利用这些规则稍微调整一下代码来仅获取词条链接，使用的正则表达式为 ^(/wiki/)((?!:).)*$")：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

html = urlopen('http://en.wikipedia.org/wiki/Kevin_Bacon')
bs = BeautifulSoup(html, 'html.parser')
for link in bs.find('div', {'id':'bodyContent'}).find_all(
    'a', href=re.compile('^(/wiki/)((?!:).)*$')):
    if 'href' in link.attrs:
        print(link.attrs['href'])
```

如果你运行以上代码，就会看到维基百科上凯文·贝肯词条里所有指向其他词条的链接。

当然，写程序来找出这个静态的维基百科词条里所有的词条链接很有趣，不过没什么实际用处。你需要让这段程序更像下面的形式。

- 一个函数 getLinks，可以用一个 /wiki/< 词条名称 > 形式的维基百科词条 URL 作为参数，然后以同样的形式返回一个列表，里面包含所有的词条 URL。
- 一个主函数，以某个起始词条为参数调用 getLinks，然后从返回的 URL 列表里随机选择一个词条链接，再次调用 getLinks，直到你主动停止程序，或者在新的页面上没有词条链接了。

完整的代码如下所示：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import datetime
import random
import re

random.seed(datetime.datetime.now())
def getLinks(articleUrl):
    html = urlopen('http://en.wikipedia.org{}'.format(articleUrl))
    bs = BeautifulSoup(html, 'html.parser')
    return bs.find('div', {'id':'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))

links = getLinks('/wiki/Kevin_Bacon')
```

```
    while len(links) > 0:
        newArticle = links[random.randint(0, len(links)-1)].attrs['href']
        print(newArticle)
        links = getLinks(newArticle)
```

导入需要的 Python 库之后，程序首先做的是用系统当前时间设置随机数生成器的种子。这样可以保证每次程序运行的时候，维基百科词条的选择都是一个全新的随机路径。

伪随机数和随机数种子

在前面的示例中，为了能够连续地随机遍历维基百科，我用 Python 的随机数生成器在每个页面上随机选择一个词条链接。但是，用随机数的时候需要格外小心。

虽然计算机很擅长做精确计算，但是它们处理随机事件时非常不靠谱。因此，随机数是一个难题。大多数随机数算法都努力生成一个呈均匀分布且难以预测的数字序列，但是在算法初始化阶段都需要提供一个随机数“种子”（random seed）。而完全相同的种子每次将生成同样的“随机”数序列，因此我将系统时间作为生成新随机数序列（和新随机词条序列）的起点。这样做会让程序运行的时候更具有随机性。

其实，Python 的伪随机数生成器用的是**梅森旋转**（Mersenne Twister）**算法**，它生成的随机数很难预测且呈均匀分布，就是有点儿耗费 CPU 资源。真正好的随机数可不便宜！

然后，程序定义 `getLinks` 函数，它接收一个 `/wiki/<词条名称>` 形式的维基百科词条 URL 作为参数，在前面加上维基百科的域名 `http://en.wikipedia.org`，再用该域名的 HTML 获得一个 `BeautifulSoup` 对象。之后，基于前面介绍过的参数，抽取一列词条链接所在的标签 `a` 并返回它们。

程序的主函数首先把起始页面 https://en.wikipedia.org/wiki/Kevin_Bacon 里的词条链接列表设置成链接标签列表（`links` 变量）。然后用一个循环，从页面中随机找一个词条链接标签并抽取 `href` 属性，打印这个页面，再把这个链接传入 `getLinks` 函数，重新获取新的链接列表。

当然，这里只是简单地构建一个从一个页面到另一个页面的爬虫，要解决“维基百科六度分隔理论”问题还需要再做一点儿工作。我们还应该存储 URL 链接数据并分析数据。关于这个问题的具体解决办法，请参考第 6 章内容。



异常处理

虽然为了简洁起见，我们在这些示例中忽略了大多数异常处理过程，但是要注意问题随时可能发生。例如，维基百科改变了 `bodyContent` 标签的名称怎么办呢？当程序尝试从该标签抽取文本时，程序会抛出 `AttributeError` 异常。

因此，这些脚本作为演示示例也许可以运行得很不错，但是要真正成为自动化产品代码，还需要增加更多的异常处理。关于异常处理的更多信息，请参考第 1 章的相关内容。

3.2 抓取整个网站

上一节，我们实现了在一个网站上随机地从一个链接跳到另一个链接。但是，如果你需要系统地为网站编目录，或者要搜索网站上的每一个页面，该怎么办？抓取整个网站，尤其是大型网站，是一个非常耗费内存资源的过程，最合适的工具就是用数据库来存储抓取结果的应用。但是，我们可以探索这些类型的应用的行为，而无须全面地运行它们。要了解更多关于使用数据库来运行这些应用的相关知识，请参考第 6 章。

深网和暗网

你可能听说过**深网**（deep Web）、**暗网**（dark Web）或**隐藏网络**（hidden Web）之类的术语，尤其是在最近的媒体中。它们是什么意思呢？

深网是 Web 的一部分，与**浅网**（surface Web）[2] 对立。浅网是互联网上搜索引擎可以抓到的那部分网络。据估计，互联网中其实约 90% 的网络都是深网。因为谷歌不能做像表单提交这类事情，也找不到那些没有直接链接到顶层域名上的网页，或者因为有 robots.txt 禁止而不能查看网站，所以浅网的数量相对深网还是比较少的。

暗网，也被称为 darknet，则完全是另一种网络[3]。它们也建立在已有的网络硬件基础上，但是使用 Tor 或者另一个客户端，带有运行在 HTTP 之上的应用协议，提供了一个信息交换的安全渠道。这类暗网页面也是可以抓取的，就像抓取其他网站一样，不过这超出了本书的范围。

和暗网不同，深网相对容易抓取。实际上，本书中的很多工具都会教你如何抓取那些 Google 网络机器人不能获取的深网信息。

那么，什么时候抓取整个网站是有用的，什么时候又是有害无益的呢？遍历整个网站的网络爬虫有许多好处。

生成网站地图

几年前，我曾经遇到过一个问题：一位重要的客户想对网站的一个重新设计方案进行效果评估，但是不想让我们公司进入他们的网站内容管理系统（CMS），也没有一个公开可用的网站地图。我就用爬虫抓取了整个网站，收集了所有的内链，然后把页面整理成他们网站实际使用的目录结构。这样，我很快找出了网站中我以前不曾留意的部分，并准确地计算出了需要重新设计多少网页，以及需要移动多少内容。

收集数据

我的另一位客户为了给一个专业的搜索平台创建一个工作原型，想收集一些文章（故事、博文、新闻文章等）。虽然这些网站的抓取不需要全面彻底，但是需要广泛（我们

注 2：参见 Alex Wright 的“Exploring a‘Deep Web’that Google Can’t Grasp”。

注 3：参考 Andy Greenberg 的“Hacker Lexicon: What is the Dark Web?”。

有意收集数据的网站不多）。于是我就创建了一个爬虫来递归地遍历每个网站，并且只收集那些文章页面上的数据。

全面彻底地抓取网站的常用方法是从一个顶级页面（比如主页）开始，然后搜索该页面上的所有内链，形成列表。之后，抓取这些链接跳转到的每一个页面，再把在每个页面上找到的链接形成新的列表，接着执行下一轮抓取。

很明显，这是一种复杂度迅速增加的情形。假如每个页面有 10 个内链，网站的深度是 5 个页面（中等规模网站的常见深度），那么如果你要抓取整个网站，一共得抓取的网页数量就是 10^5，即 100 000 个页面。不过，虽然“5 个页面的深度，每个页面 10 个内链”是网站的主流配置，但其实很少有网站真的有 100 000 个或更多的页面。这是因为大部分内链都是重复的。

为了避免一个页面被抓取两次，链接去重是非常重要的。在代码运行时，要把已发现的所有链接都放到一起，并保存在方便查询的集合（set）里。**集合**与列表类似，但是集合中的元素没有特定的顺序，集合只存储唯一的元素，这正是我们需要的功能。只有“新”链接才应被抓取，并从其页面中搜索其他链接：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

pages = set()
def getLinks(pageUrl):
    global pages
    html = urlopen('http://en.wikipedia.org{}'.format(pageUrl))
    bs = BeautifulSoup(html, 'html.parser')
    for link in bs.find_all('a', href=re.compile('^(/wiki/)')):
        if 'href' in link.attrs:
            if link.attrs['href'] not in pages:
                #We have encountered a new page
                newPage = link.attrs['href']
                print(newPage)
                pages.add(newPage)
                getLinks(newPage)
getLinks('')
```

为了全面地展示这个网页抓取示例是如何工作的，我降低了在前面例子里使用的“只寻找内链”的标准，不再限制爬虫抓取的页面范围，只要遇到页面就查找所有以 /wiki/ 开头的链接，也不考虑链接是不是包含冒号。提示：词条链接不包含冒号，而文档上传页面、讨论页面之类的页面 URL 都包含冒号。

一开始，用 `getLinks` 处理一个空 URL，其实就是维基百科的主页，因为在函数里空 URL 就是 `http://en.wikipedia.org`。然后，遍历首页上的每个链接，并检查它是否已经在**全局变量集合** `pages`（已经抓取的页面集合）里面了。如果不在，就添加到集合中，并打印到

屏幕上，再用 getLinks 递归地处理这个链接。



关于递归的警告

这个警告在软件开发类图书里很少提到，但是我觉得你应该注意：如果递归运行的次数非常多，前面的递归程序很可能会崩溃。

Python 默认的递归限制（程序递归地调用自身的次数）是 1000 次。因为维基百科的链接网络浩如烟海，所以这个程序达到递归限制后就会停止，除非你设置一个较大的递归计数器，或者采用其他手段不让它停止。

对于那些链接深度小于 1000 的“扁平”网站，这种方法通常可行，但有一些罕见的例外。例如，我曾经遇到过一个网站，该网站根据当前网页的地址生成新的 URL 链接。这就导致了像 blogs/blogs.../blogs/blog-post.php 这样不断重复的路径。

但是大多数时候，这种递归的技巧对于你碰到的任何典型网站都是适用的。

收集整个网站的数据

当然，如果只是从一个页面跳到另一个页面，那么网络爬虫是非常无聊的。为了有效地使用它们，在用爬虫的时候我们需要在页面上做些事情。让我们看看如何创建一个爬虫来收集页面标题、正文的第一个段落，以及编辑页面的链接（如果有的话）这些信息。

和往常一样，决定如何做好这些事情的第一步就是先观察网站上的一些页面，然后拟定一个抓取模式。通过观察几个维基百科页面，包括词条页面和非词条页面，比如隐私策略页面，就会得出下面的规则。

- 所有的标题（所有页面上，不论是词条页面、编辑历史页面还是其他页面）都是在 h1 → span 标签里，而且页面上只有一个 h1 标签。
- 前面提到过，所有的正文文本都在 div#bodyContent 标签里。但是，如果我们只想获取第一段文字，可能用 div#mw-content-text → p 更好（只选择第一段的标签）。这个规则对所有内容页面都适用，除了文件页面（例如，https://en.wikipedia.org/wiki/File:Orbit_of_274301_Wikipedia.svg），它们不包含内容文本（content text）部分。
- 编辑链接只出现在词条页面上。如果有编辑链接，都位于 li#ca-edit 标签的 li#ca-edit → span → a 里面。

调整前面的代码，我们就可以建立一个爬虫和数据收集（至少是数据打印）的组合程序：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

pages = set()
```

```
def getLinks(pageUrl):
    global pages
    html = urlopen('http://en.wikipedia.org{}'.format(pageUrl))
    bs = BeautifulSoup(html, 'html.parser')
    try:
        print(bs.h1.get_text())
        print(bs.find(id ='mw-content-text').find_all('p')[0])
        print(bs.find(id='ca-edit').find('span')
             .find('a').attrs['href'])
    except AttributeError:
        print("页面缺少一些属性！不过不用担心！")

    for link in bs.find_all('a', href=re.compile('^(/wiki/)')):
        if 'href' in link.attrs:
            if link.attrs['href'] not in pages:
                # 我们遇到了新页面
                newPage = link.attrs['href']
                print('-'*20)
                print(newPage)
                pages.add(newPage)
                getLinks(newPage)
getLinks('')
```

这个程序中的 for 循环和原来的抓取程序中基本上是一样的（除了打印一条虚线来分离不同的页面内容之外）。

因为我们不可能确保每个页面上都有所有类型的数据，所以每个打印语句都是按照数据在页面上出现的可能性从高到低排列的。也就是说，<h1> 标题标签会出现在每个页面上，所以我们首先试着获取该数据。文本内容会出现在大多数页面上（除了文件页面），因此是第二个获取的数据。“编辑”按钮只出现在标题和文本内容都已存在的页面上，但不是所有这类页面上都有“编辑”按钮，所以我们最后打印这类数据。



不同模式应对不同需求

很显然，在一个异常处理语句中包裹多行代码是有危险的。首先，你无法识别出究竟是哪行代码抛出了异常。其次，如果出于某种原因，某个页面没有标题，却有“编辑”按钮，那么由于前面已经发生异常，后面的“编辑”按钮链接就不会出现。但是，这种按照网站上信息出现的可能性高低进行排序的方法对许多网站都是可行的，偶尔会丢失一点儿数据，只要保存详细的日志就不是什么问题了。

你可能注意到了，在到目前为止的所有例子中，我们都没有“收集”那些“打印”出来的数据。显然，命令行里显示的数据是很难进一步处理的。我们将在第 5 章继续介绍信息存储和数据库创建的相关内容。

处理重定向

重定向使得 Web 服务器可以将一个域名或者 URL 指向不同位置的内容。重定向有两种类型：

- 服务器端重定向，在页面加载之前 URL 就会发生改变；
- 客户端重定向，有时我们可以看到“页面将在 10 秒钟内跳转”这类消息，这里页面在跳转到新页面之前已经加载。

对于服务器端重定向，你通常不需要担心。如果你使用的是 Python 3.x 的 `urllib` 库的话，它可以自动处理重定向问题！如果你使用的是 `requests` 库的话，需要将允许重定向的标志设置为 `True`：

```
r = requests.get('http://github.com', allow_redirects=True)
```

需要注意的是，有时候你抓取的页面的 URL 可能不是你进入该页面的 URL。

更多关于（通过 JavaScript 或者 HTML 实现的）客户端重定向的信息，可以参考第 12 章。

3.3 在互联网上抓取

每次我做有关网页抓取的演讲，总会有人问我：“你怎么创建谷歌网站？”我的回答通常会包含两点：“首先，你得有几十亿美元，买得起世界上最大的数据仓库，并把它们隐秘地放在世界各地。其次，你得写一个网络爬虫。”

谷歌在 1994 年成立的时候，就是两名斯坦福大学毕业生使用一台陈旧的服务器和一个 Python 网络爬虫。既然你已经知道如何抓取网页了，那么你已经正式拥有了成为下一个科技亿万富翁所需的工具了！

严肃地说，网络爬虫驱动着许多现代 Web 技术，你不一定需要一个大型数据仓库来使用它们。要实现任何跨站的数据分析，你确实需要构建出可以解析并存储互联网上无数网页中的数据的爬虫。

就像前一个例子一样，你将创建的网络爬虫也是顺着链接从一个页面跳到另一个页面，绘制出一张 Web 地图。但是这一次，它们不再忽略外链，而是跟着外链跳转。



不知前方水深浅

下一节的代码可以到达互联网的**任何位置**。如果我们从“维基百科六度分隔理论”中学到了什么，那便是完全有可能从一个像芝麻街那样的网站，经过几跳就到达某个非主流网站。

如果读者是小朋友，请在运行代码前咨询一下爸妈。如果法律或者宗教禁止你浏览某个网站的内容，那么你可以阅读代码示例，但是运行代码的时候请格外小心。

在你编写爬虫跟随外链跳转之前，请问自己几个问题。

- 我要收集哪些数据？数据收集可以通过抓取几个预定义的网站（永远是最简单的做法）完成吗？或者我的爬虫需要能够发现那些我可能不知道的网站吗？
- 当我的爬虫到达某个网站，它是立即顺着下一个出站链接跳到一个新网站，还是在网站上停留一会儿，深入抓取网站的内容？
- 有没有我不想抓取的一类网站？我对非英文网站的内容感兴趣吗？
- 如果我的网络爬虫引起了某个网站管理员的怀疑，我如何避免承担法律责任？（关于这个问题的更多信息，请参考第 18 章。）

将几个灵活的 Python 函数组合起来就可以实现不同类型的网络爬虫，用不超过 60 行代码就可轻松地写出来：

```
from urllib.request import urlopen
from urllib.parse import urlparse
from bs4 import BeautifulSoup
import re
import datetime
import random

pages = set()
random.seed(datetime.datetime.now())

# 获取页面中所有内链的列表
def getInternalLinks(bs, includeUrl):
    includeUrl = '{}://{}'.format(urlparse(includeUrl).scheme,
        urlparse(includeUrl).netloc)
    internalLinks = []
    # 找出所有以"/"开头的链接
    for link in bs.find_all('a',
        href=re.compile('^(/|.*'+includeUrl+')')):
        if link.attrs['href'] is not None:
            if link.attrs['href'] not in internalLinks:
                if(link.attrs['href'].startswith('/')):
                    internalLinks.append(
                        includeUrl+link.attrs['href'])
                else:
                    internalLinks.append(link.attrs['href'])
    return internalLinks

# 获取页面中所有外链的列表
def getExternalLinks(bs, excludeUrl):
    externalLinks = []
    # 找出所有以"http"或"www"开头且不包含当前URL的链接
    for link in bs.find_all('a',
        href=re.compile('^(http|www)((?!'+excludeUrl+').)*$')):
        if link.attrs['href'] is not None:
            if link.attrs['href'] not in externalLinks:
                externalLinks.append(link.attrs['href'])
    return externalLinks
```

```
def getRandomExternalLink(startingPage):
    html = urlopen(startingPage)
    bs = BeautifulSoup(html, 'html.parser')
    externalLinks = getExternalLinks(bs,
        urlparse(startingPage).netloc)
    if len(externalLinks) == 0:
        print('No external links, looking around the site for one')
        domain = '{}://{}'.format(urlparse(startingPage).scheme,
            urlparse(startingPage).netloc)
        internalLinks = getInternalLinks(bs, domain)
        return getRandomExternalLink(internalLinks[random.randint(0,
                                    len(internalLinks)-1)])
    else:
        return externalLinks[random.randint(0, len(externalLinks)-1)]

def followExternalOnly(startingSite):
    externalLink = getRandomExternalLink(startingSite)
    print('Random external link is: {}'.format(externalLink))
    followExternalOnly(externalLink)
followExternalOnly('http://oreilly.com')
```

上面这个程序从 http://oreilly.com 开始，随机地从一个外链跳到另一个外链。输出的结果如下所示：

```
http://igniteshow.com/
http://feeds.feedburner.com/oreilly/news
http://hire.jobvite.com/CompanyJobs/Careers.aspx?c=q319
http://makerfaire.com/
```

在网站首页上并不总是能发现外链。这时，为了找到外链，就需要用一种类似于前面例子中使用的抓取方法的方法，递归地深入一个网站，直到找到一个外链为止。

图 3-1 把程序操作可视化成了一个流程图。

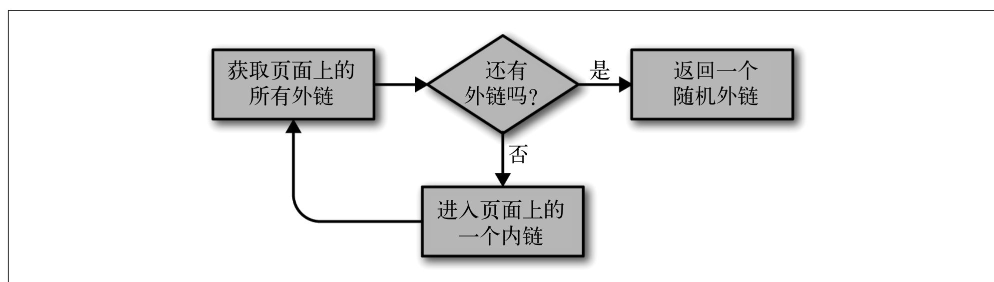


图 3-1：从互联网的网站上抓取外链的程序流程图



不要把示例程序放进产品代码

为了节省空间以及保证可读性，书中的示例程序不一定包含真实产品代码中必须有的检查和异常处理。例如，如果在抓取的网站里一个外链都没有找到（虽然不太可能，但是如果程序运行的时候够长，总会遇到这种情况），程序会一直运行，直到达到 Python 的递归限制为止。

一种增强爬虫稳健性的方法，是将其和第 1 章中介绍的处理网络连接异常的代码结合起来。这样，当出现 HTTP 错误或者服务器异常时，代码就可以选择一个不同的 URL。

在以任何严肃的目的运行代码之前，请确认你已经在可能出现问题的地方都放置了检查语句。

把任务分解成像“获取页面上所有外链”这样的小函数的好处是，以后可以方便地重构代码，以满足另一个抓取任务的需求。例如，如果你的目标是抓取一个网站中所有的外链并且逐一记录下来，你可以增加下面的函数：

```
#收集在网站上发现的所有外链列表
allExtLinks = set()
allIntLinks = set()

def getAllExternalLinks(siteUrl):
    html = urlopen(siteUrl)
    domain = '{}://{}'.format(urlparse(siteUrl).scheme,
        urlparse(siteUrl).netloc)
    bs = BeautifulSoup(html, 'html.parser')
    internalLinks = getInternalLinks(bs, domain)
    externalLinks = getExternalLinks(bs, domain)

    for link in externalLinks:
        if link not in allExtLinks:
            allExtLinks.add(link)
            print(link)
    for link in internalLinks:
        if link not in allIntLinks:
            allIntLinks.add(link)
            getAllExternalLinks(link)

allIntLinks.add('http://oreilly.com')
getAllExternalLinks('http://oreilly.com')
```

可以将这段代码视为共同协作的两个循环，一个是收集内链，一个是收集外链。程序的流程如图 3-2 所示。

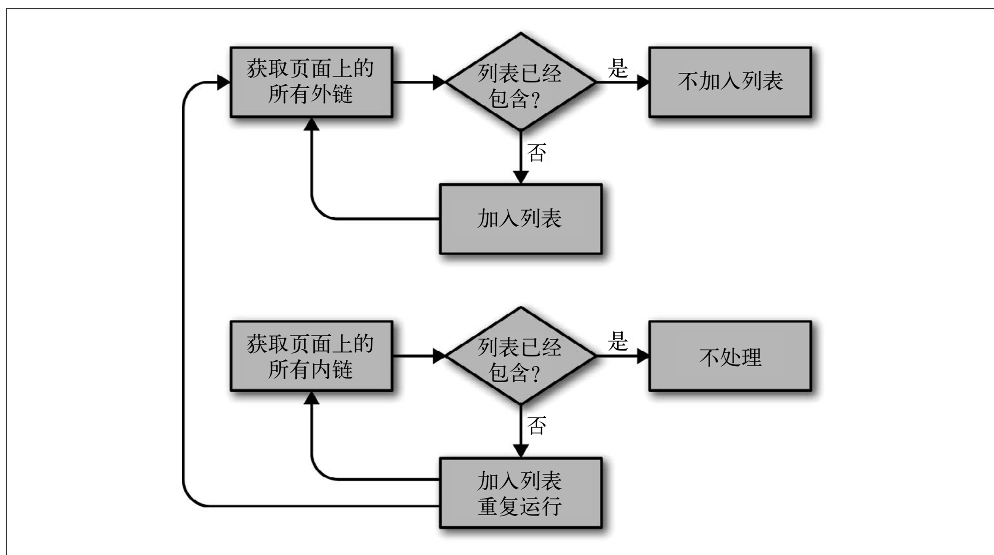


图 3-2：收集内链和外链的程序流程图

写代码之前拟个大纲或画个流程图是个很好的编程习惯，这么做不仅可以为后期处理节省很多时间，更重要的是可以防止自己在爬虫变得越来越复杂时乱了分寸。

第4章

网络爬虫模型

即使你能掌控数据和输入，编写干净、可扩展的代码也是很难的。而编写网络爬虫代码时，需要抓取并存储来自多组网站的各种各样的数据，而且这些数据是程序员无法控制的，这就带来了独特的组织挑战。

你可能被要求从多个网站抓取新闻文章或者博客文章，而这些网站的模板和布局各不相同。一个网站的 `h1` 标签包含文章的标题，另外一个网站的 `h1` 标签包含网站本身的标题，而文章的标题则在 `<span id="title">` 中。

你可能需要灵活地控制要抓取哪些网站以及如何抓取，还需要一种在不需要编写很多代码的情况下，尽可能快地添加新网站或者修改已有网站的方法。

你可能被要求从不同的网站抓取产品价格，最终实现对同一产品的价格比较。可能这些价格是不同货币形式的，可能你还需要将这些数据和来自某种非 Web 来源的外部数据合并起来。

尽管网络爬虫的应用几乎是无止尽的，但大型、可扩展的爬虫往往分为几种模式。通过学习这些模式并识别它们适用的情境，你可以大幅改善你的网络爬虫的可维护性和稳健性。

本章重点介绍从各种网站收集少数几种“类型”的数据（例如餐馆评论、新闻、公司资料），并将这些数据类型存储为从数据库读写的 Python 对象的网络爬虫。

4.1 规划和定义对象

网页抓取过程中的一个常见陷阱，是完全基于眼前可见的内容定义自己希望抓取的数据。例如，如果你想抓取产品数据，可能首先查看服装店并且确定你抓取的每种产品需要具有

以下字段：

- 产品名称
- 价格
- 描述
- 尺寸
- 颜色
- 面料类型
- 顾客评分

而查看另外一个网站时，你发现该网站的网页上列出了 SKU 值（库存单元，用于跟踪和预订商品）。你希望同时抓取这个数据，即便该数据并未出现在你抓取的第一个网站上！于是你加上这个字段：

- SKU 项

尽管服装可能是一个不错的开始，但你还希望该爬虫可以扩展到其他类型的产品上。你开始浏览其他网站的产品部分，并且确定你还需要抓取以下信息：

- 精装 / 平装
- 哑光印刷 / 光面印刷
- 顾客评价数量
- 生产商的链接

很明显，这是一种不可持续的方法。每次在一个网站上看到一条新信息就给你的产品类型添加属性，这会导致需要跟踪太多的字段。不仅如此，每次你抓取一个新的网站，都得对该网站拥有的字段以及你已经积累的字段做详尽的分析，以增加新的字段（即更改 Python 对象类型以及数据库结构）。这样得到的结果可能是一个杂乱的、很难读的数据集，从而造成使用困难。

当决定抓取哪些数据时，最好的做法是忽视所有的网站。当你启动一个可扩展的大型项目时，不是首先查看单个网站并且问“存在什么？”，而是要自问“我需要什么？”，然后想方设法从中寻找所需信息。

可能你真正需要做的就是比较多个商店的产品价格，并且追踪这些价格的变化。这种情况下，你需要足够的信息来唯一地识别各个产品，就是这么简单。

- 产品名称
- 制造商
- 产品 ID 号（如果可以获得或者相关的话）

值得注意的一点是，以上这些信息并不特定于某一商店。例如，产品评论、评分、价格，甚至描述都是针对特定商店的特定产品的。这些信息可以单独保存。

其他信息（产品的颜色、材质）是特定于产品的，但是可能很稀疏，因为并不是所有产品都有这个信息。所以我们需要后退一步，对你考虑的每一项都做一个清单检查，然后问自己以下几个问题。

- 这个信息可以帮助项目实现目标吗？如果我不包括该信息，是否会造成阻碍？还是说该信息有了固然好，但是并不会影响任何结果？
- 如果该信息将来**可能**有帮助，但是我并不确定，那么晚些时候再抓取会有多大的困难？
- 这个数据对于我已经抓取的信息来说是否冗余？
- 将数据存储在这个对象中是否符合逻辑？（正如前面提到的，如果同一产品在不同网站上的描述不一致的话，那么存储该产品的描述信息就没有意义。）

如果你确定需要抓取该数据，那么就要问自己以下问题，然后确定如何在代码中存储并处理这些数据。

- 该数据是稀疏的还是密集的？它与每个清单都相关并且会出现在其中，还是只与部分清单相关？
- 该数据有多大？
- 在数据较大的情况下，我每次运行分析时都需要检索该数据，还是只是偶尔需要使用该数据？
- 这种类型的数据有多大的变化性？我需要经常加入新的属性、修改类型（例如面料样式可能是经常修改的属性）吗？还是说该数据一直保持不变（鞋的码数）？

假如你计划对产品属性和价格做一些元数据分析，例如书的页码，或者布的面料，或者将来一些与价格相关的其他属性。你一一探查上述问题，发现数据是稀疏的（仅有少数产品有这些属性之一），因此你可能决定经常增加或者移除部分属性。这样的话，创建一个如下所示的产品类型可能比较合理：

- 产品名称
- 制造商
- 产品 ID（如果可以获得 / 相关）
- 属性（可选列表或字典）

属性类型如下所示：

- 属性名称
- 属性值

这样，你就可以灵活地添加新的产品属性，而不需要重新设计数据模式或者重写代码。当

决定好如何在数据库中存储这些属性后，你可以在 JSON 中编写 `attribute` 字段，或者将每个属性与产品 ID 一起存在一个单独的表格中。关于实现这些类型的数据库模型的更多信息，可以查看第 6 章。

你也可以将上述问题用于其他需要存储的数据。为了跟踪每个产品的价格，你可能需要以下字段：

- 产品 ID
- 商店 ID
- 价格
- 价格的日期 / 时间戳

但是如果产品的属性修改了产品的价格，怎么办？例如，商店中大衬衫的价格可能比小衬衫高，因为大衬衫需要更多的劳动或材料。在这种情况下，你可以考虑将每个尺码的衬衫产品拆分成单独的产品列表（这样每个衬衫产品可以单独定价），或者创建一个新的项目类型来存储产品实例的相关信息，并包含以下字段：

- 产品 ID
- 实例类型（这里是衬衫的尺码）

而每个价格应该如下所示：

- 产品实例 ID
- 商店 ID
- 价格
- 价格的日期 / 时间戳

尽管这里“产品和价格”这个主题可能看起来过于具体，但你需要问自己的基本问题，以及设计 Python 对象时的逻辑，几乎适用于所有情境。

如果你抓取新闻文章，可能需要以下基本信息：

- 标题
- 作者
- 日期
- 内容

但是有些文章还包含“修改日期”“相关文章”或者“社交媒体分享次数”。你需要这些信息吗？这些信息和你的项目相关吗？如果不是所有的新闻网站都使用所有形式的社交媒体，并且社交媒体网站可能随着时间的推移变得更加流行或不再流行，那么你如何高效而灵活地存储社交媒体分享次数？

当面临一个新的项目时，很容易立马开始写 Python 代码来抓取网站。而数据模型通常是后面考虑的内容，并且通常会被你抓取的第一个网站的数据可用性和数据格式所影响。

但是数据模型是所有代码的基础。模型中糟糕的决定很容易导致代码编写和维护的问题，或者导致难以抽取和高效地使用数据。特别是当你处理很多类型的网站（包括已知的和未知的）时，认真思考并规划你究竟需要抓取什么以及如何存储变得非常关键。

4.2　处理不同的网站布局

类似 Google 这样的搜索引擎，最大的优点之一就是能够从大量的网站中抽取相关和有用的数据，而不需要具备关于网站结构本身的知识。尽管我们人类可以立刻识别出页面的标题和主要内容（设计非常糟糕的网站除外），但机器完成这项任务却非常困难。

幸运的是，在大多数网页抓取任务中，你不会去抓取你从未见过的网站，而是从一些人为预选的网站中抓取。这就意味着你不需要使用复杂的算法或者机器学习去识别页面上的哪段文字看起来"最像标题"或者可能是"主要内容"。你可以手动确定网页上的各个元素。

最显而易见的方法是，为每个网站单独编写一个网络爬虫或者页面解析器。每个爬虫或解析器以一个 URL、字符串或者 `BeautifulSoup` 对象作为输入，并返回一个抓取的 Python 对象。

以下是一个 `Content` 类的示例（代表网站上的一块内容，如新闻文章），其中两个抓取器函数以 `BeautifulSoup` 对象作为输入，返回一个 `Content` 实例：

```
import requests

class Content:
    def __init__(self, url, title, body):
        self.url = url
        self.title = title
        self.body = body

def getPage(url):
    req = requests.get(url)
    return BeautifulSoup(req.text, 'html.parser')

def scrapeNYTimes(url):
    bs = getPage(url)
    title = bs.find("h1").text
    lines = bs.find_all("p", {"class":"story-content"})
    body = '\n'.join([line.text for line in lines])
    return Content(url, title, body)

def scrapeBrookings(url):
    bs = getPage(url)
    title = bs.find("h1").text
    body = bs.find("div",{"class","post-body"}).text
    return Content(url, title, body)
```

```
url = 'https://www.brookings.edu/blog/future-development'
    '/2018/01/26/delivering-inclusive-urban-access-3-unc'
    'omfortable-truths/'
content = scrapeBrookings(url)
print('Title: {}'.format(content.title))
print('URL: {}\n'.format(content.url))
print(content.body)

url = 'https://www.nytimes.com/2018/01/25/opinion/sunday/'
    'silicon-valley-immortality.html"
content = scrapeNYTimes(url)
print('Title: {}'.format(content.title))
print('URL: {}\n'.format(content.url))
print(content.body)
```

当你为额外的新闻网站添加抓取器函数时，可能会发现存在一种模式。每个网站的解析函数基本上在做同样的事情：

- 选择标题元素并从标题中抽取文本
- 选择文章的主要内容
- 按需选择其他内容项
- 返回此前由字符串实例化的 `Content` 对象

这里唯一与网站相关的变量是用于获取信息的 CSS 选择器。BeautifulSoup 的 `find` 和 `find_all` 函数需要两个输入参数——一个标签字符串和一个带有键 / 值属性的字典，这样你可以传递这两个参数来定义网站本身的结构以及目标数据的位置。

为了更简便，你可以不处理所有的标签参数和键 / 值对，而是用单个 CSS 选择器使用 BeautifulSoup 的 `select` 函数选定你希望抓取的信息，并且将这些选择器放入到一个字典对象中。

```
class Content:
    """
    所有文章/网页的共同基类
    """

    def __init__(self, url, title, body):
        self.url = url
        self.title = title
        self.body = body

    def print(self):
        """
        用灵活的打印函数控制结果
        """
        print("URL: {}".format(self.url))
        print("TITLE: {}".format(self.title))
        print("BODY:\n{}".format(self.body))
```

```
class Website:
    """
    描述网站结构的信息
    """

    def __init__(self, name, url, titleTag, bodyTag):
        self.name = name
        self.url = url
        self.titleTag = titleTag
        self.bodyTag = bodyTag
```

注意，这里 `Website` 类并不存储任何从页面本身抓取的信息，而是存储关于**如何**抓取数据的指令。它也不存储“My Page Title”这样的标题。它只会存储字符串标签 `h1`，表明了在哪里可以找到标题。这就是这个类被命名为 `Website`（它包含适用于整个网站的信息）而不是 `Content`（它包含来自单个网页的信息）的原因。

使用这些 `Content` 类和 `Website` 类，你就可以编写一个 `Crawler` 去抓取任何网站的任何网页的标题和内容：

```
import requests
from bs4 import BeautifulSoup

class Crawler:

    def getPage(self, url):
        try:
            req = requests.get(url)
        except requests.exceptions.RequestException:
            return None
        return BeautifulSoup(req.text, 'html.parser')

    def safeGet(self, pageObj, selector):
        """
        用于从一个BeautifulSoup对象和一个选择器获取内容的辅助函数。
        如果选择器没有找到对象，就返回空字符串
        """
        selectedElems = pageObj.select(selector)
        if selectedElems is not None and len(selectedElems) > 0:
            return '\n'.join(
            [elem.get_text() for elem in selectedElems])
        return ''

    def parse(self, site, url):
        """
        从指定URL提取内容
        """
        bs = self.getPage(url)
        if bs is not None:
            title = self.safeGet(bs, site.titleTag)
            body = self.safeGet(bs, site.bodyTag)
```

```
            if title != '' and body != '':
                content = Content(url, title, body)
                content.print()
```

以下代码定义了网站对象并开启了流程：

```
crawler = Crawler()

siteData = [
    ['O\'Reilly Media', 'http://oreilly.com',
    'h1', 'section#product-description'],
    ['Reuters', 'http://reuters.com', 'h1',
    'div.StandardArticleBody_body_1gnLA'],
    ['Brookings', 'http://www.brookings.edu',
    'h1', 'div.post-body'],
    ['New York Times', 'http://nytimes.com',
    'h1', 'p.story-content']
]
websites = []
for row in siteData:
    websites.append(Website(row[0], row[1], row[2], row[3]))

crawler.parse(websites[0], 'http://shop.oreilly.com/product/'\
    '0636920028154.do')
crawler.parse(websites[1], 'http://www.reuters.com/article/'\
    'us-usa-epa-pruitt-idUSKBN19W2D0')
crawler.parse(websites[2], 'https://www.brookings.edu/blog/'\
    'techtank/2016/03/01/idea-to-retire-old-methods-of-policy-education/')
crawler.parse(websites[3], 'https://www.nytimes.com/2018/01/'\
    '28/business/energy-environment/oil-boom.html')
```

尽管这个方法乍看起来并不比为每个新的网站编写一个新的 Python 函数简单多少，但是想象一下，如果你的系统从抓取 4 个网站变成抓取 20 个甚至 200 个网站，会发生什么。

每个字符串列表写起来相对容易，并且也不会占用太多空间。它可以通过一个数据库或者 CSV 文件加载。它可以从远程源导入，或者交给一个有前端经验的非程序员来填充并加入新的网站，而他们并不需要阅读代码。

当然，不足之处是你牺牲了一定的灵活性。在第一个例子中，每个网站都有自己的函数来选择和解析 HTML，以获取最终结果。在第二个例子中，每个网站必须具有一定的结构，即特定的字段必须存在，从字段取出的数据必须干净，并且每个目标字段必须有唯一且可靠的 CSS 选择器。

但是我相信这种方法的强大功能和灵活性足以弥补其缺陷。下一节将介绍这一基本模板的具体应用和扩展，这样你就可以处理缺失的字段，抓取不同类型的数据，仅抓取网站的特定部分，以及存储更复杂的页面信息。

4.3 结构化爬虫

如果你还需要手动定位到想抓取的每个链接，那么创建灵活和可修改的网站布局类型并不会带来多大的好处。上一章介绍了自动抓取网站和发现新页面的各种方式。

这一节将介绍如何将这些方法应用于结构良好的、可扩展的网站爬虫，以自动搜集链接和发现数据。本节将展示 3 种基本的网络爬虫结构，我认为它们可以应用于大多数情形，不过你在抓取网站时可能需要做一些改动。如果你碰到了特殊情形，遇到了抓取问题，我也希望你能借鉴这些结构来设计优雅、健壮的爬虫。

4.3.1 通过搜索抓取网站

抓取网站的一种最简单的方法是像人类一样使用搜索条。尽管在网站上搜索关键词或者主题并收集搜索结果的过程，看起来是一个随着网站的不同而有很大可变性的任务，但有几个关键点使得这个任务出人意料地容易。

- 大多数网站通过将主题作为参数在 URL 中传递，来获得特定主题的搜索结果列表。例如，`http://example.com?search=myTopic`。这个 URL 的第一部分可以存为 `Website` 对象的一个属性，简单地在其后添加主题。
- 在搜索后，大多数网站以非常好识别的链接列表的形式呈现结果页面，通常会使用一个如 `<span class="result">` 的标签，而其准确的形式也可以存为 `Website` 对象的一个属性。
- 每个**结果链接**要么是一个相对 URL（例如 /articles/page.html），要么是一个绝对 URL（例如 http://example.com/articles/page.html）。不管是相对 URL 还是绝对 URL，你都可以将其存为 `Website` 对象的一个属性。
- 当定位到并规范化搜索页面的 URL 后，你就成功地将问题简化为上一节示例中的问题了——抽取给定格式网站的数据。

我们接下来用代码实现该算法。`Content` 类和前面例子中的基本一样。你需要添加 URL 属性来跟踪发现内容的位置：

```
class Content:
    """所有文章/网页的共同基类"""

    def __init__(self, topic, url, title, body):
        self.topic = topic
        self.title = title
        self.body = body
        self.url = url

    def print(self):
        """
        用灵活的打印函数控制结果
        """
        print("New article found for topic: {}".format(self.topic))
```

```
        print("TITLE: {}".format(self.title))
        print("BODY:\n{}".format(self.body))
        print("URL: {}".format(self.url))
```

程序中的 `Website` 类加入了一些新的属性。如果你附加了要搜索的主题，那么 `searchUrl` 定义了可以在哪里获得搜索结果。`resultListing` 定义了存放每个结果信息的“盒子”（box），而 `resultUrl` 定义了这个盒子中的标签，这些标签即为结果的准确 URL。`absoluteUrl` 属性是一个布尔值，它表示搜索结果是绝对 URL 还是相对 URL。

```
class Website:
    """描述网站结构的信息"""

    def __init__(self, name, url, searchUrl, resultListing,
        resultUrl, absoluteUrl, titleTag, bodyTag):
        self.name = name
        self.url = url
        self.searchUrl = searchUrl
        self.resultListing = resultListing
        self.resultUrl = resultUrl
        self.absoluteUrl=absoluteUrl
        self.titleTag = titleTag
        self.bodyTag = bodyTag
```

crawler.py 被进一步扩展，它包含 `Website` 数据、待搜索的主题列表和两个对所有这些网站和主题进行迭代的循环。它还包括一个 `search` 函数，该函数对特定网站和主题的搜索页面进行导航，并抽取页面中所有的结果 URL。

```
import requests
from bs4 import BeautifulSoup

class Crawler:

    def getPage(self, url):
        try:
            req = requests.get(url)
        except requests.exceptions.RequestException:
            return None
        return BeautifulSoup(req.text, 'html.parser')

    def safeGet(self, pageObj, selector):
        childObj = pageObj.select(selector)
        if childObj is not None and len(childObj) > 0:
            return childObj[0].get_text()
        return ""

    def search(self, topic, site):
        """
        根据主题搜索网站并记录所有找到的页面
        """
        bs = self.getPage(site.searchUrl + topic)
        searchResults = bs.select(site.resultListing)
        for result in searchResults:
```

```
                url = result.select(site.resultUrl)[0].attrs["href"]
                # 检查一下是相对URL还是绝对URL
                if(site.absoluteUrl):
                    bs = self.getPage(url)
                else:
                    bs = self.getPage(site.url + url)
                if bs is None:
                    print("Something was wrong with that page or URL. Skipping!")
                    return
                title = self.safeGet(bs, site.titleTag)
                body = self.safeGet(bs, site.bodyTag)
                if title != '' and body != '':
                    content = Content(topic, title, body, url)
                    content.print()


crawler = Crawler()

siteData = [
    ['O\'Reilly Media', 'http://oreilly.com',
        'https://ssearch.oreilly.com/?q=','article.product-result',
        'p.title a', True, 'h1', 'section#product-description'],
    ['Reuters', 'http://reuters.com',
        'http://www.reuters.com/search/news?blob=',
        'div.search-result-content','h3.search-result-title a',
        False, 'h1', 'div.StandardArticleBody_body_1gnLA'],
    ['Brookings', 'http://www.brookings.edu',
        'https://www.brookings.edu/search/?s=',
        'div.list-content article', 'h4.title a', True, 'h1',
        'div.post-body']
]
sites = []
for row in siteData:
    sites.append(Website(row[0], row[1], row[2],
                         row[3], row[4], row[5], row[6], row[7]))
topics = ['python', 'data science']
for topic in topics:
    print("GETTING INFO ABOUT: " + topic)
    for targetSite in sites:
        crawler.search(topic, targetSite)
```

上述代码会对 topics 列表中的所有主题进行循环，并且在开始抓取每个主题之前会先做声明：

```
GETTING INFO ABOUT python
```

然后对 sites 列表中的所有网站进行循环，抓取每个主题的每个特定网站。每次成功地抓取了页面的信息后，它都会在控制台打印如下信息：

```
New article found for topic: python
URL: http://example.com/examplepage.html
TITLE: Page Title Here
BODY: Body content is here
```

注意，这里程序是对所有的主题进行循环，然后在内循环中对所有的网站进行循环。那么为什么不调换位置呢，即从一个网站抓取所有主题后，再从下一个网站抓取所有主题？先对所有主题进行迭代，对任何 Web 服务器来说都是一种均衡分配负载的做法。如果你有上百个主题和几十个网站，这一点尤其重要。你不会对一个网站一次性发起上万个请求，而是发起 10 个请求，等几分钟后再发起 10 个请求，然后再等几分钟，如此反复。

尽管两种做法最终的请求次数是一样的，但通常还是最好将这些请求合理地分配在不同的时间。要达到这个目的，一种简单的办法是多思考如何构建你的循环。

4.3.2　通过链接抓取网站

上一章介绍了在网页中识别内链和外链，然后利用这些链接抓取网站的一些方法。本节，你将会学习把这些基本方法融合到一个更灵活的网站爬虫中，该爬虫可以跟踪任意遵循特定 URL 模式的链接。

这种爬虫非常适用于从一个网站抓取所有数据的项目，而不适用于从特定搜索结果或页面列表抓取数据的项目。它还非常适用于网站页面组织得很糟糕或者非常分散的情况。

这些类型的爬虫并不需要像上一节通过搜索页面进行抓取中采用的定位链接的结构化方法，因此在 `Website` 对象中不需要包含描述搜索页面的属性。但是由于爬虫并不知道待寻找的链接的位置，所以你需要一些规则来告诉它选择哪种页面。你可以用 `targetPattern`（目标 URL 的正则表达式）和布尔变量 `absoluteUrl` 来达成这一目标：

```
class Website:
    def __init__(self, name, url, targetPattern, absoluteUrl,
        titleTag, bodyTag):
        self.name = name
        self.url = url
        self.targetPattern = targetPattern
        self.absoluteUrl=absoluteUrl
        self.titleTag = titleTag
        self.bodyTag = bodyTag

class Content:
    def __init__(self, url, title, body):
        self.url = url
        self.title = title
        self.body = body

    def print(self):
        print("URL: {}".format(self.url))
        print("TITLE: {}".format(self.title))
        print("BODY:\n{}".format(self.body))
```

`Content` 类和第一个爬虫例子中使用的是一样的。

`Crawler` 类从每个网站的主页开始，定位内链，并解析在每个内链页面发现的内容：

```
import re

class Crawler:
    def __init__(self, site):
        self.site = site
        self.visited = []

    def getPage(self, url):
        try:
            req = requests.get(url)
        except requests.exceptions.RequestException:
            return None
        return BeautifulSoup(req.text, 'html.parser')

    def safeGet(self, pageObj, selector):
        selectedElems = pageObj.select(selector)
        if selectedElems is not None and len(selectedElems) > 0:
            return '\n'.join([elem.get_text() for
                elem in selectedElems])
        return ''

    def parse(self, url):
        bs = self.getPage(url)
        if bs is not None:
            title = self.safeGet(bs, self.site.titleTag)
            body = self.safeGet(bs, self.site.bodyTag)
            if title != '' and body != '':
                content = Content(url, title, body)
                content.print()

    def crawl(self):
        """
        获取网站主页的页面链接
        """
        bs = self.getPage(self.site.url)
        targetPages = bs.findAll('a',
            href=re.compile(self.site.targetPattern))
        for targetPage in targetPages:
            targetPage = targetPage.attrs['href']
            if targetPage not in self.visited:
                self.visited.append(targetPage)
                if not self.site.absoluteUrl:
                    targetPage = '{}{}'.format(self.site.url, targetPage)
                self.parse(targetPage)

reuters = Website('Reuters', 'https://www.reuters.com', '^(/article/)', False,
    'h1', 'div.StandardArticleBody_body_1gnLA')
crawler = Crawler(reuters)
crawler.crawl()
```

与前面的例子相比，这里的另外一个变化是：Website 对象（在这个例子中是变量 reuters）是 Crawler 对象本身的一个属性。这样做的作用是将已访问过的页面存储在爬虫中，但是也意味着必须针对每个网站实例化一个新的爬虫，而不是重用一个爬虫去抓取网站列表。

不管你是选择一个与网站无关的爬虫，还是将网站作为爬虫的一个属性，这都是一个需要根据自身需求进行权衡的决定。两种方法在功能实现上都是没有问题的。

另外需要注意的是，这个爬虫会从主页开始抓取，但是在所有页面都被记录后，就不会继续抓取了。你可能希望编写一个爬虫，将第 3 章中介绍的某种模式融合进来，然后查看所访问的每个页面中更多的目标 URL。你甚至还可以跟踪每个页面中涉及的所有 URL（不仅仅是匹配目标模式的 URL），然后查看这些 URL 是否包含目标模式。

4.3.3 抓取多种类型的页面

与抓取预定义好的页面集合不同，抓取一个网站的所有内链会带来一个挑战，即你不知道会获得什么。好在有几种基本的方法可以识别页面类型。

通过URL

一个网站中所有的博客文章可能都会包含一个 URL（例如 http://example.com/blog/title-of-post）。

通过网站中存在或者缺失的特定字段

如果一个页面包含日期，但是不包含作者名字，那你可以将其归类为新闻稿。如果它有标题、主图片、价格，但是没有主要内容，那么它可能是一个产品页面。

通过页面中出现的特定标签识别页面

即使不抓取某个标签内的数据，你仍然可以利用这个标签。你的爬虫可以寻找类似于 `<div id="related-products">` 这样的元素来识别产品页面，即便是爬虫对相关产品的内容并不感兴趣。

为了跟踪多个页面类型，你需要在 Python 中有多个类型的页面对象。这通过两种方式来实现。

如果页面都是相似的（它们基本上都是相同类型的内容），你可能需要在现有的网页对象中加入一个 `pageType` 属性：

```
class Website:
    """所有文章/网页的共同基类"""

    def __init__(self, type, name, url, searchUrl, resultListing,
        resultUrl, absoluteUrl, titleTag, bodyTag):
        self.name = name
        self.url = url
        self.titleTag = titleTag
        self.bodyTag = bodyTag
        self.pageType = pageType
```

如果你在一个类 SQL 的数据库中对这些页面进行排序，这种模式类型意味着这些页面应该被存放在同一张表中，并且加入一个额外的 `pageType` 列。

如果你抓取的页面或内容各不相同（它们包含不同类型的字段），就需要为每个页面类型创建一个新的对象。当然，有些东西是所有网页共有的——它们都有一个 URL，也可能都有一个名称或者页面标题。这种情况非常适合用子类：

```
class Website:
    """所有文章/网页的共同基类"""

    def __init__(self, name, url, titleTag):
        self.name = name
        self.url = url
        self.titleTag = titleTag
```

这不是一个由你的爬虫直接使用的对象，而是将被你的页面类型引用的对象：

```
class Product(Website):
    """产品页面要抓取的信息"""
    def __init__(self, name, url, titleTag, productNumber, price):
        Website.__init__(self, name, url, TitleTag)
        self.productNumberTag = productNumberTag
        self.priceTag = priceTag

class Article(Website):
    """文章页面要抓取的信息"""
    def __init__(self, name, url, titleTag, bodyTag, dateTag):
        Website.__init__(self, name, url, titleTag)
        self.bodyTag = bodyTag
        self.dateTag = dateTag
```


这个产品页面扩展了 Website 基类，并且加入了仅适用于产品的 productNumber 和 price 属性，而 Article 类加入了 body 和 date 属性，这两个属性是不适用于产品的。

你可以用这两个类去抓取一个商店网站，该网站除了产品，可能还包含博客文章或新闻稿。

4.4 关于网络爬虫模型的思考

从互联网抓取信息犹如从消防水带中饮水。互联网上有很多东西，你需要什么或者你如何需要它并非总是清晰的。对于任何大型网页抓取项目（甚至是一些小项目）来说，第一步都是回答这些问题。

当抓取来自多个源或者多个域的相似数据时，你的目标应该总是将其规范化。处理带有相同和可比较的字段的数据，总是比处理完全依赖于其源格式的数据容易得多。

在很多情况下，构建爬虫时应该假定未来会加入更多的数据源，目标是减少加入这些新数据源所带来的开销。即使某个网站乍看起来并不适用于你的模型，但也可能会有一些细节证明它确实是适用的。从长远看，能够识别潜藏的模式可以为你节约时间、金钱，也能避免很多烦恼。

数据间的联系也不应该被忽视。你是否在所有数据源中寻找带有“类型”“尺码”或“主题”等属性的信息？你如何存储、检索并将这些属性概念化？

软件架构是一个广泛而重要的主题，需要在整个职业生涯中逐渐掌握。幸运的是，网页抓取的软件架构是一套有限且可管理的技能，并且很容易掌握。在继续抓取数据的过程中，你可能会发现同样的基本模式反复地出现。创建一个具有良好结构的网络爬虫不需要具备很多晦涩难懂的知识，但是确实需要你后退一步仔细思考你的项目。

第5章

Scrapy

上一章介绍了一些创建大型、可扩展、（最重要的！）易维护的网络爬虫的技术和模式。虽然手动创建非常简单，但是许多现成的库、框架甚至带图形界面的工具可以代劳，使用它们至少可以让生活轻松点儿。

本章将介绍网络爬虫开发中一个最好的框架：Scrapy。在我写本书第一版的时候，Scrapy 还没有发布针对 Python 3.x 的版本，因此我在书里只为它安排了一节内容。后来，这个库不断升级，目前已经支持 Python 3.3 以上版本，而且还添加了一些新功能，因此我非常想把那一节内容扩展成一章。

写网络爬虫的一个挑战是经常需要重复同样的任务：找出网页中的所有链接，评估内链与外链的差异，再跳转到新的网页。虽然掌握这些基本模式很有用，也便于从零开始创建爬虫，但是 Scrapy 可以帮你搞定里面的诸多细节。

当然，Scrapy 并不能揣测你的心思。你仍然需要定义网页模板，告诉它开始抓取的位置，为你要找的网页定义 URL 模式。但是在这些场景中，它都提供了一个整洁的框架来帮你组织代码。

5.1 安装Scrapy

Scrapy 不仅提供了从其网站进行下载的工具，也提供了用第三方安装管理器（如 pip）安装 Scrapy 的指令。

由于 Scrapy 比较大也比较复杂，它通常不是一个可用如下传统方式安装的框架。

```
$ pip install Scrapy
```

我之所以说“通常”，是因为尽管理论上可以通过 pip 安装成功，但是我经常在安装过程中遇到复杂的依赖问题、版本不匹配问题，以及其他无法解决的 bug。

如果你执意要用 pip 安装 Scrapy，那么强烈推荐你用虚拟环境（关于虚拟环境的更多细节，请参考 1.2.1 节中的“用虚拟环境保存库文件”）。

我更喜欢用 Anaconda 包管理器进行安装。Anaconda 是 Continuum 公司开发的产品，主要是为了方便人们搜索和安装流行的 Python 数据科学包。它管理的许多 Python 包都会在后面的章节中用到，例如 NumPy 和 NLTK。

Anaconda 安装完成后，通过下面的命令就可以安装 Scrapy：

```
conda install -c conda-forge scrapy
```

如果你遇到了问题，或者需要最新信息，请参阅 Scrapy 官方文档中的安装指南。

蜘蛛初始化

当 Scrapy 框架安装完成之后，还需要为每一个蜘蛛（spider）做一些配置。一个**蜘蛛**就是一个 Scrapy 项目，和它的名称一样，就是用来爬网（抓取网页）的。我在这一章都用“蜘蛛”特指 Scrapy 项目，而用“爬虫”（crawler）表示“任意用或不用 Scrapy 抓取网页的程序”。

如果在当前目录中创建新的蜘蛛，就运行下面的命令：

```
$ scrapy startproject wikiSpider
```

这行命令会在项目所在的目录中创建一个新的子目录，名为 wikiSpider。目录里面的文件结构如下所示：

- scrapy.cfg
- wikiSpider
  — spiders
    — __init.py__
  — items.py
  — middlewares.py
  — pipelines.py
  — settings.py
  — __init.py__

这些 Python 文件都用桩代码进行初始化，为用户提供一种创建新蜘蛛项目的快捷方式。本章中的每一节内容都是围绕 wikiSpider 项目展开的。

5.2 创建一个简易爬虫

为了创建一个爬虫，首先需要在 spiders 文件夹里面增加一个新文件，路径为 wikiSpider/wikiSpider/spiders/article.py。在你刚刚创建的 article.py 文件中，写上以下代码：

```
import scrapy

class ArticleSpider(scrapy.Spider):
    name='article'

    def start_requests(self):
        urls = [
            'http://en.wikipedia.org/wiki/Python_'
            '%28programming_language%29',
            'https://en.wikipedia.org/wiki/Functional_programming',
            'https://en.wikipedia.org/wiki/Monty_Python']
        return [scrapy.Request(url=url, callback=self.parse)
            for url in urls]

    def parse(self, response):
        url = response.url
        title = response.css('h1::text').extract_first()
        print('URL is: {}'.format(url))
        print('Title is: {}'.format(title))
```

这个类的名称（`ArticleSpider`）和文件名（wikiSpider）不一样，这表明这个类在 wikiSpider 的众多类目中专门用于抓取文章网页，后面你可能还需要用它搜索其他类型的网页。

在处理包含多种内容的大型网站时，你可能需要为每种内容（像博客文章、新闻稿、文章等）分配不同的 Scrapy item，每个具有不同的字段，但它们都在同一个 Scrapy 项目中运行。项目里面的每个蜘蛛的名称必须唯一。

关于这个蜘蛛还需要注意的是两个函数 `start_requests` 和 `parse`。

`start_requests` 函数是 Scrapy 定义的程序入口，用于生成 Scrapy 用来抓取网站的 `Request` 对象。

`parse` 是一个用户定义的回调函数，通过 `callback=self.parse` 传递给 `Request` 对象。在后面的内容中，你会看到 `parse` 函数更强大的功能，不过现在只是让它打印网页的标题。

你可以进入 wikiSpider/wikiSpider/spiders 文件夹，然后运行 `article` 蜘蛛：

```
$ scrapy runspider article.py
```

Scrapy 默认的输出结果非常啰嗦。除了一堆调试信息之外，打印的结果可能像下面这样：

```
2018-01-21 23:28:57 [scrapy.core.engine] DEBUG: Crawled (200)
<GET https://en.wikipedia.org/robots.txt> (referer: None)
2018-01-21 23:28:57 [scrapy.downloadermiddlewares.redirect]
DEBUG: Redirecting (301) to <GET https://en.wikipedia.org/wiki/
Python_%28programming_language%29> from <GET http://en.wikipedia.org/
wiki/Python_%28programming_language%29>
2018-01-21 23:28:57 [scrapy.core.engine] DEBUG: Crawled (200)
<GET https://en.wikipedia.org/wiki/Functional_programming>
(referer: None)
URL is: https://en.wikipedia.org/wiki/Functional_programming
Title is: Functional programming
2018-01-21 23:28:57 [scrapy.core.engine] DEBUG: Crawled (200)
<GET https://en.wikipedia.org/wiki/Monty_Python> (referer: None)
URL is: https://en.wikipedia.org/wiki/Monty_Python
Title is: Monty Python
```

这个爬虫到达了 `start_urls` 列表中的 3 个网页，收集信息，然后停止。

5.3 带规则的抓取

上一节的蜘蛛还不能算是真正意义的爬虫，只是抓取了一组网页的 URL 而已。它还不具备独立寻找新网页的能力。为了让它成为一只功能齐全的爬虫，你还需要用 Scrapy 的 `CrawlSpider` 类来完善它。



用 GitHub 仓库组织代码

Scrapy 框架不方便直接在 Jupyter notebook 中运行，所以像前几章那样渐进式增加代码难以实现。为了展示所有的示例代码，上一节的爬虫保存在 article.py 文件里，而接下来的示例（创建一个 Scrapy 蜘蛛遍历多个网页）保存在 articles.py（注意这里用复数形式）文件里。

后面的示例都会保存在单独的文件中，每一节都会出现新的文件名。运行这些示例时，请确保使用了正确的文件名。

下面的类在 GitHub 仓库的 articles.py 文件中可以找到：

```
from scrapy.contrib.linkextractors import LinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule

class ArticleSpider(CrawlSpider):
    name = 'articles'
    allowed_domains = ['wikipedia.org']
    start_urls = ['https://en.wikipedia.org/wiki/'
        'Benevolent_dictator_for_life']
    rules = [Rule(LinkExtractor(allow=r'.*'), callback='parse_items',
        follow=True)]
```

```
    def parse_items(self, response):
        url = response.url
        title = response.css('h1::text').extract_first()
        text = response.xpath('//div[@id="mw-content-text"]//text()')
            .extract()
        lastUpdated = response.css('li#footer-info-lastmod::text')
            .extract_first()
        lastUpdated = lastUpdated.replace(
            'This page was last edited on ', '')
        print('URL is: {}'.format(url))
        print('title is: {} '.format(title))
        print('text is: {}'.format(text))
        print('Last updated: {}'.format(lastUpdated))
```

新的 `ArticleSpider` 扩展了 `CrawlSpider` 类。它没有使用 `start_requests` 函数，而是定义了两个列表 `start_urls` 和 `allowed_domains`。这是为了告诉蜘蛛从哪开始抓取，以及哪些域名的链接应该保留，哪些应该忽略。

另外还定义了一个 `rules` 列表，为哪些链接应该保留、哪些应该忽略提供了进一步的说明（在本示例中，用正则表达式 `.*` 保留了所有链接）。

除了提取每个网页的标题和 URL，蜘蛛还增加了一些新的 item。每个网页的文字内容都是通过 XPath 选择器提取的。XPath 通常用于获取包含子标签（例如，一段文本里的标签 `<a>`）的文字内容。如果你用 CSS 选择器处理，那么子标签里面的所有文字都会被忽略。

另外，网页最后更新的日期字符串也从页脚解析出来，保存在 `lastUpdated` 变量中。

现在你可以到 wikiSpider/wikiSpider/spiders 文件夹里运行示例了，代码如下：

```
$ scrapy runspider articles.py
```



警告：这个蜘蛛会一直运行

虽然这个蜘蛛和前面那个一样在命令行中运行，但是它不会终止（至少很长一段时间内不会终止），除非你用 Ctrl-C 或者关闭终端来强行终止它。考虑到维基百科的服务器负载，请不要长时间运行它。

蜘蛛运行的时候会遍历 wikipedia.org，然后保留所有含 wikipedia.org 域名的链接，打印网页的标题，并忽略所有外链：

```
2018-01-21 01:30:36 [scrapy.spidermiddlewares.offsite]
DEBUG: Filtered offsite request to 'www.chicagomag.com':
<GET http://www.chicagomag.com/Chicago-Magazine/June-2009/
Street-Wise/>
2018-01-21 01:30:36 [scrapy.downloadermiddlewares.robotstxt]
DEBUG: Forbidden by robots.txt: <GET https://en.wikipedia.org/w/
index.php?title=Adrian_Holovaty&action=edit&section=3>
title is: Ruby on Rails
```

```
URL is: https://en.wikipedia.org/wiki/Ruby_on_Rails
text is: ['Not to be confused with ', 'Ruby (programming language)',
 '.', '\n', '\n', 'Ruby on Rails', ... ]
Last updated:  9 January 2018, at 10:32.
```

这已经是一个非常好的爬虫了，但是它可以使用一些限制。它不只是访问维基百科的文章网页时，也会在非文章网页上“漫步”，例如：

```
title is: Wikipedia:General disclaimer
```

让我们仔细看看这行代码，采用 Scrapy 的 `Rule` 和 `LinkExtractor`：

```
rules = [Rule(LinkExtractor(allow=r'.*'), callback='parse_items',
    follow=True)]
```

这行代码提供了 Scrapy 的 `Rule` 对象列表，这些对象定义了所有链接的过滤规则。当设置多个规则时，每个链接都要按顺序检查。匹配的第一个规则用来决定如何处理链接。如果链接不能匹配任何规则，就会被忽略。

一个 `Rule` 共包含 6 个参数。

`link_extractor`
: 这是唯一的必选参数，是一个 `LinkExtractor` 对象。

`callback`
: 用来解析网页内容的函数。

`cb_kwargs`
: 一个要传入回调函数的参数字典。这个字典的形式是 `{arg_name1: arg_value1, arg_name2: arg_value2}`，在重用同样的解析函数处理稍微不同的任务时，会是一个很方便的工具。

`follow`
: 设置是否需要将当前页面中找到的链接添加到后面的抓取里。如果没有提供回调函数，那么默认值为 `True`（毕竟，如果你没有对网页做任何处理，那么显然你至少还想用它来继续抓取网站）。如果提供了回调函数，则这个参数的默认值是 `False`。

`LinkExtractor` 是一个简单的类，专门用于根据提供的规则，识别和返回 HTML 内容页面中的链接。它有许多参数可用来根据 CSS 和 XPath 选择器、标签（你可以在锚标签之外寻找链接！）、域名等属性接受或拒绝链接。

`LinkExtractor` 类是可以扩展的，可以增加自定义参数。关于链接提取器的更多信息，请参考 Scrapy 的文档。

尽管 LinkExtractor 类有很多灵活的特性，但是常用的参数只有两个。

allow

允许匹配正则表达式的所有链接。

deny

拒绝匹配正则表达式的所有链接。

在单个解析函数中用两个独立的 Rule 和 LinkExtractor 类，你可以创建一个蜘蛛来抓取维基百科，识别所有的文章网页和带标签的非文章网页（articlesMoreRules.py）：

```
from scrapy.contrib.linkextractors import LinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule

class ArticleSpider(CrawlSpider):
    name = 'articles'
    allowed_domains = ['wikipedia.org']
    start_urls = ['https://en.wikipedia.org/wiki/'
        'Benevolent_dictator_for_life']
    rules = [
        Rule(LinkExtractor(allow='^(/wiki/)((?!:).)*$'),
            callback='parse_items', follow=True,
            cb_kwargs={'is_article': True}),
        Rule(LinkExtractor(allow='.*'), callback='parse_items',
            cb_kwargs={'is_article': False})
    ]

    def parse_items(self, response, is_article):
        print(response.url)
        title = response.css('h1::text').extract_first()
        if is_article:
            url = response.url
            text = response.xpath('//div[@id="mw-content-text"]'
                '//text()').extract()
            lastUpdated = response.css('li#footer-info-lastmod'
                '::text').extract_first()
            lastUpdated = lastUpdated.replace('This page was '
                'last edited on ', '')
            print('Title is: {} '.format(title))
            print('title is: {} '.format(title))
            print('text is: {}'.format(text))
        else:
            print('This is not an article: {}'.format(title))
```

前面说过，每个链接都会按照列表中的所有规则进行过滤。所有文章网页（以 /wiki/ 开始而且不包含冒号）都会先在 parse_items 函数中处理，默认参数是 is_article=True。然后所有的非文章链接都会传入到 parse_items 函数中，该函数的参数是 is_article=False。

当然，如果你只想收集文章类型网页并忽略其他页面，用这个方法可能不太合适。更容易的方法是直接忽略那些不匹配 URL 模式的网页，同时排除第二条规则（和 is_article 变

量）。然而，这种方法可能更适合处理特殊场景：URL 信息或者在抓取过程中收集到的信息会影响网页的解析方式。

5.4　创建item

到目前为止，你已经看到了许多用 Scrapy 查找、解析和抓取网站的方法，但是 Scrapy 还提供了许多有用的工具，可帮助你组织已收集的 item，并将它们保存到带有明确定义的字段的自定义对象中。

为了帮助你组织所收集的信息，你需要创建一个 `Article` 对象。在 items.py 文件中创建一个新的名为 `Article` 的 item。

当你打开 items.py 文件时，它应该像这样：

```
# -*- coding: utf-8 -*-

# 在此为抓取的item定义模型
#
# 参见文档：
# http://doc.scrapy.org/en/latest/topics/items.html

import scrapy

class WikispiderItem(scrapy.Item):
    # 在此定义item字段：
    # name = scrapy.Field()
    pass
```

把默认的 `Item` 示例代码替换成新的 `Article` 类，它扩展了 `scrapy.Item`：

```
import scrapy

class Article(scrapy.Item):
    url = scrapy.Field()
    title = scrapy.Field()
    text = scrapy.Field()
    lastUpdated = scrapy.Field()
```

你定义了从每个网页收集的 3 个字段：标题、URL 和页面最后的更新日期。

如果你要从多种网页类型中收集数据，那么你应该在 items.py 中为每种类型定义单独的类。如果你的 item 非常大，或者你开始向你的 item 对象中添加更多的解析功能，那么你可能还希望将每个 item 提取到独立的文件中。然而，当 item 比较小的时候，我还是喜欢把它们都放在一个文件中。

请注意在 articleItems.py 文件中，为了创建新的 `Article` item，需要对 `ArticleSpider` 类做一些调整：

```
from scrapy.contrib.linkextractors import LinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule
from wikiSpider.items import Article

class ArticleSpider(CrawlSpider):
    name = 'articleItems'
    allowed_domains = ['wikipedia.org']
    start_urls = ['https://en.wikipedia.org/wiki/Benevolent'
        '_dictator_for_life']
    rules = [
        Rule(LinkExtractor(allow='(/wiki/)((?!:).)*$'),
            callback='parse_items', follow=True),
    ]

    def parse_items(self, response):
        article = Article()
        article['url'] = response.url
        article['title'] = response.css('h1::text').extract_first()
        article['text'] = response.xpath('//div[@id='
            '"mw-content-text"]//text()').extract()
        lastUpdated = response.css('li#footer-info-lastmod::text')
            .extract_first()
        article['lastUpdated'] = lastUpdated.replace('This page was '
            'last edited on ', '')
        return article
```

如果用下面的命令运行这个文件：

```
$ scrapy runspider articleItems.py
```

屏幕就会以 Python 字典的形式输出 Scrapy 调试信息以及每篇文章的 item：

```
2018-01-21 22:52:38 [scrapy.spidermiddlewares.offsite] DEBUG:
Filtered offsite request to 'wikimediafoundation.org':
<GET https://wikimediafoundation.org/wiki/Terms_of_Use>
2018-01-21 22:52:38 [scrapy.core.engine] DEBUG: Crawled (200)
<GET https://en.wikipedia.org/wiki/Benevolent_dictator_for_life
#mw-head> (referer: https://en.wikipedia.org/wiki/Benevolent_
dictator_for_life)
2018-01-21 22:52:38 [scrapy.core.scraper] DEBUG: Scraped from
<200 https://en.wikipedia.org/wiki/Benevolent_dictator_for_life>
{'lastUpdated': ' 13 December 2017, at 09:26.',
'text': ['For the political term, see ',
          'Benevolent dictatorship',
          '.',
          ...
```

使用 Scrapy 的 `Items` 并不只是为了良好地组织代码，或者让结果的可读性更好。`Items` 提供了许多输出和处理数据的工具，后面会介绍。

5.5　输出item

Scrapy 用 `Item` 对象确定它需要从浏览的网页中保留哪些信息。Scrapy 可以将信息保存为不同的格式，例如 CSV、JSON 和 XML 文件，这可以用下面的命令实现：

```
$ scrapy runspider articleItems.py -o articles.csv -t csv
$ scrapy runspider articleItems.py -o articles.json -t json
$ scrapy runspider articleItems.py -o articles.xml -t xml
```

每次运行 `articleItems` 爬虫时，结果都会以指定的格式写入所提供的文件中。如果文件不存在，就会自动创建。

你可能已经注意到，在前面示例中创建的 `articles` 蜘蛛里面，`text` 变量是一个字符串列表，而不是一个单独的字符串。列表中的每个字符串表示一个 HTML 元素内的文字，但是在 `<div id="mwcontent-text">` 里面收集到的文字其实是由许多子元素构成的。

Scrapy 可以很好地处理这类复杂场景。例如，在 CSV 文件格式中，它会把列表转换为字符串，并对所有逗号进行转义，从而保证一个列表的所有文字都保存在 CSV 的一个字段中。

在 XML 文件里，列表的每个元素都被保存在子标签中：

```
<items>
<item>
    <url>https://en.wikipedia.org/wiki/Benevolent_dictator_for_life</url>
    <title>Benevolent dictator for life</title>
    <text>
        <value>For the political term, see </value>
        <value>Benevolent dictatorship</value>
        ...
    </text>
    <lastUpdated> 13 December 2017, at 09:26.</lastUpdated>
</item>
....
```

在 JSON 格式中，列表仍然被保存为列表。

当然，你可以自己使用 `Item` 对象，按照你想要的方式将它们写入文件或数据库，只需在爬虫的解析函数里增加适当的代码即可。

5.6　item管线组件

虽然 Scrapy 是单线程的，但是它能够异步发出和处理多个请求。这样它就会比本书前面介绍的爬虫的速度快，尽管我一直坚信在网页抓取中快不一定就更好。

由于你要抓取的网站的 Web 服务器必须处理每一个请求，因此作为一个好公民，你必须评估这种服务器锤击（server hammering）行为是否有必要（甚至是否明智，因为许多网站也

有能力和意愿阻止它们所认为的恶意抓取行为)。关于网页抓取的道德问题和适当限制爬虫的重要性，请参考第 18 章。

用 Scrapy 的 item 管线组件可以进一步提升网页抓取的速度，因为可以在等待请求返回结果的过程中完成所有数据处理，而不是等待数据处理完成后再发起新请求。在数据处理需要大量时间时，以及在计算密集型的任务中，这种优化有时是不可或缺的。

为了创建一个 item 管线组件，需要用到本章一开始创建的 settings.py 文件。你应该会看到里面被注释的几行代码：

```
# 配置item管线组件
# 参见https://doc.scrapy.org/en/latest/topics/item-pipeline.html
#ITEM_PIPELINES = {
#    'wikiSpider.pipelines.WikispiderPipeline': 300,
#}
```

将后面 3 行代码注释去掉，替换成下面的代码：

```
ITEM_PIPELINES = {
    'wikiSpider.pipelines.WikispiderPipeline': 300,
}
```

这里提供了一个 Python 类 `wikiSpider.pipelines.WikispiderPipeline`，用来处理数据；还提供了一个整数，用于表示当存在多个数据处理类时，运行管线组件的顺序。虽然使用任何整数都可以，但是通常使用介于 0 到 1000 之间的整数，管线组件按照顺序方式运行。

现在你需要增加管线组件类，并重写原来的蜘蛛，这样就可以在蜘蛛收集数据的同时，由管线组件承担数据处理的重任。可以在你原来的蜘蛛里写一个 `parse_items` 方法来返回响应，并让管线组件创建 `Article`：

```
def parse_items(self, response):
    return response
```

然而，Scrapy 框架不允许这么做，必须返回一个 `Item` 对象（例如扩展了 `Item` 类的 `Article`）。因此 `parse_items` 现在的任务就是提取原始数据，尽可能少做数据处理，然后传递给管线组件：

```
from scrapy.contrib.linkextractors import LinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule
from wikiSpider.items import Article

class ArticleSpider(CrawlSpider):
    name = 'articlePipelines'
    allowed_domains = ['wikipedia.org']
    start_urls = ['https://en.wikipedia.org/wiki/Benevolent_dictator_for_life']
    rules = [
        Rule(LinkExtractor(allow='(/wiki/)((?!:).)*$'),
```

```
                callback='parse_items', follow=True),
        ]

        def parse_items(self, response):
            article = Article()
            article['url'] = response.url
            article['title'] = response.css('h1::text').extract_first()
            article['text'] = response.xpath('//div[@id='
                '"mw-content-text"]//text()').extract()
            article['lastUpdated'] = response.css('li#'
                'footer-info-lastmod::text').extract_first()
            return article
```

这个文件在 GitHub 仓库中被保存为 articlePipelines.py。

当然，现在还需要增加管线组件，将 settings.py 文件和升级的蜘蛛连结起来。当 Scrapy 项目在首次初始化时，会创建一个 wikiSpider/wikiSpider/pipelines.py 文件：

```
# -*- coding: utf-8 -*-

# 在此定义你的item管线组件
#
# 别忘了将管线组件添加到ITEM_PIPELINES设置
# 参见https://doc.scrapy.org/en/latest/topics/item-pipeline.html


class WikispiderPipeline(object):
    def process_item(self, item, spider):
        return item
```

这个示例类应该替换成你的新管线组件代码。在前面的几节中，你已经收集了两个原始格式的字段，而这些可能需要进行额外的数据处理：`lastUpdated`（一个表示日期的、格式糟糕的字符串对象）和 `text`（一个混乱的由字符串片段组成的数组）。

用下面的代码来替换 wikiSpider/wikiSpider/pipelines.py 文件中的示例代码：

```
from datetime import datetime
from wikiSpider.items import Article
from string import whitespace

class WikispiderPipeline(object):
    def process_item(self, article, spider):
        dateStr = article['lastUpdated']
        article['lastUpdated'] = article['lastUpdated']
            .replace('This page was last edited on', '')
        article['lastUpdated'] = article['lastUpdated'].strip()
        article['lastUpdated'] = datetime.strptime(
            article['lastUpdated'], '%d %B %Y, at %H:%M.')
        article['text'] = [line for line in article['text']
            if line not in whitespace]
        article['text'] = ''.join(article['text'])
        return article
```

WikispiderPipeline 类里面有一个 process_item 方法，它将 Article 对象作为参数，将 lastUpdated 字符串解析成 Python 的 datetime 对象，而且对 text 字符串进行清理并将数组合并成一个字符串。

对于每一个管线组件类来说，process_item 是一个必选方法。Scrapy 用这个方法异步处理蜘蛛收集到的 Items。如果你像上一节那样将结果输出到 JSON 或 CSV 文件中，返回的经过解析的 Article 对象会被记录到 Scrapy 的日志或者打印出来。

现在你可以在两个地方处理数据：一个是蜘蛛中的 parse_items 方法，另一个是管线组件中的 process_item 方法。

可以在 settings.py 文件里声明处理不同任务的多个管线组件。然而，Scrapy 会把所有 item（无论何种类型）按照顺序传递给每一个管线组件。在数据到达管线组件之前，面向具体 item 的数据解析可能在蜘蛛里面完成更合适。不过如果解析需要耗费很长时间，那么你可以需要考虑将数据处理移动到管线组件中（在那里可以异步处理），并且增加一个 item 类型的过滤：

```
def process_item(self, item, spider):
    if isinstance(item, Article):
        # 面向具体Article类型的数据解析
```

在编写 Scrapy 项目，尤其是大型项目时，做哪些数据处理以及在哪里进行这些处理是需要仔细考虑的重头戏。

5.7 Scrapy日志管理

虽然 Scrapy 生成的调试信息很有用，但是你可能会觉得它非常啰嗦。其实你可以轻松地调整日志的等级，只要在 Scrapy 项目的 settings.py 文件里增加一行代码即可：

```
LOG_LEVEL = 'ERROR'
```

Scrapy 用的是标准日志等级制度，如下所示：

- CRITICAL（关键）
- ERROR（错误）
- WARNING（警告）
- DEBUG（调试）
- INFO（信息）

如果将日志等级设置为 ERROR，那么只有 CRITICAL 和 ERROR 日志会显示。如果日志等级设置为 INFO，那么所有等级的日志都会显示，以此类推。

除了通过 settings.py 文件控制日志等级，还可以通过命令行参数控制日志。如果要将日志输出到一个单独的日志文件中，而不显示在终端里，可以用下面的命令行定义一个日志文件：

```
$ scrapy crawl articles -s LOG_FILE=wiki.log
```

如果之前没有创建日志文件，它会在当前目录中创建一个新的日志文件，并将所有日志输出到该文件里，这样你的终端就会很干净，只显示你手动添加的 Python 打印语句。

5.8 更多资源

Scrapy 是一个非常强大的工具，可以处理许多网页抓取方面的问题。它可以自动抓取 URL 并和预定义的规则进行比较，确保所有的 URL 是唯一的，并且根据需要将相对路径的 URL 转换为全链接，还可以递归爬行到更深的页面里。

对于 Scrapy 的能力，本章涉及的只是皮毛，我希望你参考 Scrapy 的文档，以及阅读 Dimitrios Kouzis-Loukas 所著的《精通 Python 爬虫框架 Scrapy》，该书对这个框架进行了详细的阐述。

Scrapy 是一个非常庞大的、具有多种功能的爬虫库。虽然它的功能可以无缝衔接，但有许多重叠区域，使得用户可以轻松地开发出符合自己特定风格的爬虫。如果你想用 Scrapy 实现什么本章没有提到的功能，那么很可能有一种或几种方法可以做到！

第 6 章
存储数据

虽然在命令行里显示运行结果很有意思，但是随着数据不断增多，需要进行数据聚合和分析时，将数据打印到命令行就不是办法了。为了可以远程使用大部分网络爬虫，你还需要把抓取到的数据存储起来。

本章将介绍 3 种主要的数据管理方法，它们对绝大多数应用都适用。如果你准备创建一个网站的后端服务或者创建自己的 API，那么可能需要让爬虫把数据写入数据库。如果你需要一个快速简单的方法收集网上的文档，然后保存到你的硬盘里，那么可能需要创建一个文件流（file stream）。如果还要为偶然事件提个醒儿，或者每天定时收集当天累计的数据，就给自己发一封邮件吧！

抛开与网页抓取的关系，大数据存储和与数据交互的能力，在现代程序开发中也已经是重中之重了。这一章的内容其实是实现第二部分许多示例的基础。如果你对自动数据存储相关的知识不太了解，我强烈建议你至少浏览一下本章内容。

6.1 媒体文件

存储媒体文件有两种主要方式：只获取文件 URL 链接，以及直接把源文件下载下来。你可以通过媒体文件所在的 URL 链接直接引用它。这样做的优点如下。

- 爬虫运行得更快，耗费的流量更少，因为只要链接，不需要下载文件。
- 可以节省很多存储空间，因为只需要存储 URL 链接就可以。
- 存储 URL 的代码更容易写，也不需要实现文件下载代码。
- 不下载文件能够降低目标主机服务器的负载。

不过这么做也有一些缺点。

- 在你的网站或应用中内嵌这些外站 URL 链接被称为**盗链**（hotlinking）。使用盗链可能会让你麻烦不断，因为每个网站都会采取防盗链措施。
- 因为你的链接文件在别人的服务器上，所以你的应用就要跟着别人的节奏运行了。
- 盗链是很容易改变的。如果你把盗链图片放在博客上，要是被对方服务器发现，很可能会被恶搞。如果你把 URL 链接存起来准备以后再存储文件，可能用的时候链接已经失效了，或者是变成了完全无关的内容。
- 现实中的 Web 浏览器不仅可以请求 HTML 页面并切换页面，也会下载访问页面上所有的资源。下载文件会让你的爬虫看起来更像是人在浏览网站，这样做反而有好处。

如果你还在犹豫究竟是存储文件，还是只存储文件的 URL 链接，可以想想这些文件是要多次使用，还是放进数据库之后就只是等着“落灰”，再也不会被打开。如果答案是后者，那么最好只存储这些文件的 URL。如果答案是前者，那么就继续往下看！

用来获取网页内容的 `urllib` 库还包含有用来获取文件内容的方法。下面的程序使用 `urllib.request.urlretrieve` 从远程 URL 下载图片：

```
from urllib.request import urlretrieve
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com')
bs = BeautifulSoup(html, 'html.parser')
imageLocation = bs.find('a', {'id': 'logo'}).find('img')['src']
urlretrieve (imageLocation, 'logo.jpg')
```

这段程序从 http://pythonscraping.com 下载 logo 图片，然后在程序运行的文件夹里保存为 logo.jpg 文件。

如果你只需要下载一个文件，而且知道如何获取它，以及它的文件类型，这么做就可以了。但是大多数爬虫都不可能一天只下载一个文件。下面的程序会把 http://pythonscraping.com 主页上所有 `src` 属性的内部文件都下载下来：

```
import os
from urllib.request import urlretrieve
from urllib.request import urlopen
from bs4 import BeautifulSoup

downloadDirectory = 'downloaded'
baseUrl = 'http://pythonscraping.com'

def getAbsoluteURL(baseUrl, source):
    if source.startswith('http://www.'):
        url = 'http://{}'.format(source[11:])
    elif source.startswith('http://'):
```

```
        url = source
    elif source.startswith('www.'):
        url = source[4:]
        url = 'http://{}'.format(source)
    else:
        url = '{}/{}'.format(baseUrl, source)
    if baseUrl not in url:
        return None
    return url

def getDownloadPath(baseUrl, absoluteUrl, downloadDirectory):
    path = absoluteUrl.replace('www.', '')
    path = path.replace(baseUrl, '')
    path = downloadDirectory+path
    directory = os.path.dirname(path)

    if not os.path.exists(directory):
        os.makedirs(directory)

    return path

html = urlopen('http://www.pythonscraping.com')
bs = BeautifulSoup(html, 'html.parser')
downloadList = bs.findAll(src=True)

for download in downloadList:
    fileUrl = getAbsoluteURL(baseUrl, download['src'])
    if fileUrl is not None:
        print(fileUrl)
        urlretrieve(fileUrl, getDownloadPath(baseUrl, fileUrl, downloadDirectory))
```



程序运行注意事项

你知道从网上下载未知文件的那些警告吗？这个程序会把页面上所有的文件下载到你的硬盘里，可能会包含一些 bash 脚本、.exe 文件，甚至可能是恶意软件。

如果你从没运行过任何下载到电脑里的文件，电脑就是安全的吗？尤其是当你用管理员权限运行这个程序时，你的电脑基本已经处于危险之中。如果你执行了网页上的一个文件，那个文件把自己传送到了 ../../../../usr/bin/python 里面，会发生什么呢？等下一次你再运行 Python 程序时，你的电脑就可能会安装恶意软件。

这个程序只是为了演示；请不要随意运行它，因为这里没有对所有下载文件的类型进行检查，也不应该用管理员权限运行它。记得经常备份重要的文件，不要在硬盘上存储敏感信息，小心驶得万年船。

这个程序首先使用 Lambda 函数（第 2 章介绍过）选择首页上所有带 `src` 属性的标签。然后对 URL 链接进行清理和标准化，获得文件的绝对路径（而且去掉了外链）。最后，每个文件都会下载到程序所在文件夹的 downloaded 文件夹里。

这里 Python 的 `os` 模块用来获取每个下载文件的目标文件夹，建立完整的路径。`os` 模块是 Python 与操作系统进行交互的接口，它可以操作文件路径，创建目录，获取运行进程和环境变量的信息，以及其他系统相关的操作。

6.2　把数据存储到CSV

CSV（comma-separated values，**逗号分隔值**）是存储表格数据的常用文件格式。Microsoft Excel 和很多应用都支持 CSV 格式，因为它很简洁。下面就是一个 CSV 文件的例子：

```
fruit,cost
apple,1.00
banana,0.30
pear,1.25
```

和 Python 一样，CSV 里空白（whitespace）也是很重要的：每一行都用一个换行符分隔，列与列之间用逗号分隔（因此得名“逗号分隔”）。CSV 文件（有时也叫**字符分隔值**文件）还可以用 Tab 字符或其他字符分隔行，但是不太常见，用得不多。

如果你只想从网页上把 CSV 文件下载到电脑里，不打算做任何解析和修改，那么这一节后面的内容就没必要看了。只要用上一节介绍的文件下载方法下载并保存为 CSV 格式就行了。

Python 的 csv 库可以非常简单地修改 CSV 文件，甚至可以从零开始创建一个 CSV 文件：

```
import csv

csvFile = open('test.csv', 'w+')
try:
    writer = csv.writer(csvFile)
    writer.writerow(('number', 'number plus 2', 'number times 2'))
    for i in range(10):
        writer.writerow( (i, i+2, i*2))
finally:
    csvFile.close()
```

这里提个醒儿：Python 新建文件的机制考虑得非常周到。如果 test.csv 不存在，Python 会自动创建该文件（不会自动创建文件夹）。如果该文件已经存在，Python 会用新的数据覆盖 test.csv 文件。

运行完成后，你会看到一个 CSV 文件：

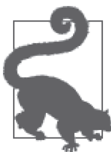
```
number,number plus 2,number times 2
0,2,0
1,3,2
2,4,4
...
```

网页抓取的一个常用功能就是获取 HTML 表格并写入 CSV 文件。维基百科的文本编辑器对比词条（https://en.wikipedia.org/wiki/Comparison_of_text_editors）中提供了一个非常复杂的 HTML 表格，其中用到了颜色、链接、排序，以及其他在写入 CSV 文件之前需要忽略的 HTML 元素。用 BeautifulSoup 和 `get_text()` 函数，你可以用十几行代码完成这件事：

```
import csv
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://en.wikipedia.org/wiki/'
    'Comparison_of_text_editors')
bs = BeautifulSoup(html, 'html.parser')
# 主对比表格是当前页面上的第一个表格
table = bs.findAll('table',{'class':'wikitable'})[0]
rows = table.findAll('tr')

csvFile = open('editors.csv', 'wt+')
writer = csv.writer(csvFile)
try:
    for row in rows:
        csvRow = []
        for cell in row.findAll(['td', 'th']):
            csvRow.append(cell.get_text())
        writer.writerow(csvRow)
finally:
    csvFile.close()
```



获取单个表格的更简便的方法

如果你有很多 HTML 表格，且每个都要转换成 CSV 文件，或者要将许多 HTML 表格汇总到一个 CSV 文件，那么可以把这个程序整合到爬虫里。但是，如果这种事情你只需要做一次，那么更好的办法是复制粘贴。选中 HTML 表格内容然后复制粘贴到 Excel 或 Google Docs 里，就可以另存为 CSV 格式，不需要写代码就能搞定！

这个程序会在程序上一层目录的 files 文件夹里生成一个 CSV 文件 ../files/editors.csv。

6.3 MySQL

MySQL 是目前最受欢迎的开源关系型数据库管理系统。一个开源项目具有如此之竞争力实在是令人意外，它的流行程度可与另外两个闭源的商业数据库系统比肩：微软的 SQL Server 和甲骨文的 Oracle 数据库。

它这么流行是不无原因的。对大多数应用来说，MySQL 都是不二选择。它是一种非常

灵活、稳定、功能齐全的 DBMS，许多顶级的网站都在用它，比如 YouTube[1]、Twitter[2] 和 Facebook[3] 等。

因为它受众广泛，免费，开箱即用，所以它也是网页抓取项目中常用的数据库，我们将在本书后面的示例中使用它。

> **"关系型"数据库？**
>
> **关系型数据**就是有关联的数据。就是这么简单！
>
> 开个玩笑！当计算机科学家说起关系型数据时，他们指的是那些并非孤立的数据——它们的属性与其他的数据是有关联的。例如，"用户 A 在学校 B 上学"，这里用户 A 在数据库的"用户"表中，而学校 B 是在数据库的"学校"表中。
>
> 在本章后面的内容里，我们将介绍数据关系的不同类型，以及如何有效地把数据存储到 MySQL（或其他关系型数据库）里。

6.3.1 安装MySQL

如果你第一次接触 MySQL，安装数据库听着可能有点儿吓人（如果你是老手，可以跳过本节）。其实，安装方法和安装其他软件一样简单。归根到底，MySQL 就是由一系列数据文件构成的，存储在你的远端服务器或本地的电脑上，里面包含了数据库存储的所有信息。MySQL 软件层提供了一种通过命令行界面与数据交互的便捷方法。例如，下面的命令会把用户表 users 中所有名字为"Ryan"的用户找出来：

```
SELECT * FROM users WHERE firstname = "Ryan"
```

如果你使用的是基于 Debian 的 Linux 发行版（或者具有 `apt-get` 的操作系统），安装 MySQL 很简单：

```
$ sudo apt-get install mysql-server
```

只要稍微留意一下安装过程，看看电脑是不是可以满足安装的内存需求，然后在安装提示的地方为 root 用户设置新密码就可以了。

macOS 和 Windows 系统上的安装过程有点儿复杂。如果你没有甲骨文账户，下载 MySQL 安装包之前需要先注册一下。

如果你用 macOS 系统，请先下载对应的安装包（http://dev.mysql.com/downloads/mysql/）。

注 1：Joab Jackson, "YouTube Scales MySQL with Go Code," PCWorld, December 15, 2012.

注 2：Jeremy Cole and Davi Arnaut, "MySQL at Twitter," The Twitter Engineering Blog, April 9, 2012.

注 3："MySQL and Database Engineering: Mark Callaghan," March 4, 2012.

选择 .dmg 安装包，登录网站或者创建一个账户，开始下载文件。下载完成后打开安装包，你会看到一个简单的安装向导（参见图 6-1）。

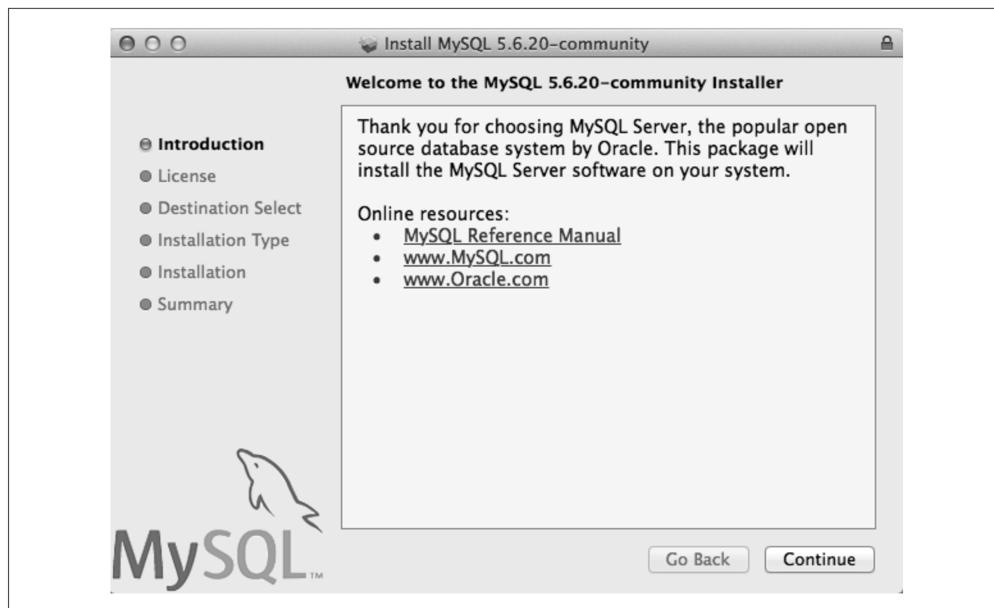


图 6-1：macOS 的 MySQL 安装工具

采用默认安装步骤就可以，本书后面都假设你采用的是默认安装步骤。

如果觉得下载安装包再运行安装工具太无聊，也可以用 macOS 的包管理器 Homebrew 安装。当 Homebrew 安装好以后，用下面的命令安装 MySQL：

```
$ brew install mysql
```

Homebrew 是一个伟大的开源工具，与 Python 包完美结合。其实，本书使用的大多数 Python 第三方库都可以用 Homebrew 安装。如果你还没用过，强烈推荐你试一下！

在 macOS 上安装好 MySQL 之后，你可以用下面的命令启动 MySQL 服务器：

```
$ cd /usr/local/mysql
$ sudo ./bin/mysqld_safe
```

在 Windows 系统上，安装和运行 MySQL 更复杂一些，但是有个方便的安装工具（http://dev.mysql.com/downloads/windows/installer/）可以简化这个过程。下载该工具后，它会引导你安装 MySQL（参见图 6-2）。



图 6-2：Windows 的 MySQL 安装工具

用默认选项安装 MySQL 就可以，不过有一个地方要注意：在 Setup Type（类型设置）页面，建议你选择“Server Only”（只选服务器）选项，这可以避免安装一堆微软的软件和库文件。然后，你就可以用默认设置安装，跟着提示一步步操作就可以启动 MySQL 服务器了。

6.3.2　基本命令

MySQL 服务器启动之后，可以通过多种方法与数据库交互。因为有很多工具具有图形界面，所以你不用 MySQL 的命令行（或者很少用命令行）也能管理数据库。像 phpMyAdmin 和 MySQL Workbench 这类工具都可以很容易地实现数据的查看、排序和插入。但是，掌握通过命令行来操作数据库依然是很重要的。

除了用户自定义变量名，MySQL 是不区分大小写的。例如，`SELECT` 和 `sElEcT` 是一样的，不过习惯上写 MySQL 语句的时候所有的 MySQL 关键词都用大写。大多数开发者还喜欢用小写字母表示数据表和数据库的名称，虽然这个标准经常不被注意。

当你首次登录 MySQL 的时候，里面是没有数据库来存放数据的。你可以创建一个：

```
> CREATE DATABASE scraping;
```

因为每个 MySQL 实例可以有多个数据库，所以使用数据库之前需要指定要使用的数据库的名称：

```
> USE scraping;
```

从现在开始（直到关闭 MySQL 链接或切换到另一个数据库），所有的命令都运行在这个新的 scraping 数据库里面。

所有操作看着都非常简单。在数据库里创建数据表应该也很简单吧？让我们在数据库里创建一个表来存储抓取的网页：

```
> CREATE TABLE pages;
```

结果显示错误：

```
ERROR 1113 (42000): A table must have at least 1 column
```

数据库可以没有表，但 MySQL 数据表必须至少有一列，否则不能创建。为了在 MySQL 里定义字段（数据列），必须在 CREATE TABLE <tablename> 语句后面，把字段的定义放进一个带括号的、内部由逗号分隔的列表中：

```
> CREATE TABLE pages (id BIGINT(7) NOT NULL AUTO_INCREMENT,
title VARCHAR(200), content VARCHAR(10000),
created TIMESTAMP DEFAULT CURRENT_TIMESTAMP, PRIMARY KEY(id));
```

每个字段定义由 3 部分组成：

- 名称（id、title、created 等）
- 数据类型（BIGINT(7)、VARCHAR、TIMESTAMP）
- 其他可选属性（NOT NULL AUTO_INCREMENT）

在字段定义列表的最后，还要定义一个**主键**（key）。MySQL 用这个主键来组织表的内容，以便于后面快速查询。本章后面将介绍如何利用这些主键以提高数据库的查询速度，但是现在，用表的 id 列作为主键就可以。

语句执行之后，你可以用 DESCRIBE 查看数据表的结构：

```
> DESCRIBE pages;
+---------+----------------+------+-----+-------------------+----------------+
| Field   | Type           | Null | Key | Default           | Extra          |
+---------+----------------+------+-----+-------------------+----------------+
| id      | bigint(7)      | NO   | PRI | NULL              | auto_increment |
| title   | varchar(200)   | YES  |     | NULL              |                |
| content | varchar(10000) | YES  |     | NULL              |                |
| created | timestamp      | NO   |     | CURRENT_TIMESTAMP |                |
+---------+----------------+------+-----+-------------------+----------------+
4 rows in set (0.00 sec)
```

当然，这还是一个空表。你可以在 pages 表里插入一些测试数据，如下所示：

```
> INSERT INTO pages (title, content) VALUES ("Test page title",
"This is some test page content. It can be up to 10,000 characters
long.");
```

需要注意的是，虽然 pages 表里有 4 个字段（id、title、content、created），但实际上你只需要插入 2 个字段（title 和 content）的数据即可。这是因为 id 字段是自动递增的（每次新插入数据行时 MySQL 自动增加 1），通常不用处理。另外，created 字段的类型是 timestamp，默认就是数据加入时的时间戳。

当然，我们也可以自定义 4 个字段的内容：

```
> INSERT INTO pages (id, title, content, created) VALUES (3,
"Test page title",
"This is some test page content. It can be up to 10,000 characters
long.", "2014-09-21 10:25:32");
```

只要你定义的整数在数据表的 id 字段里没有，它就可以顺利插入数据表。但是，这么做非常不好；除非万不得已（比如程序中断漏了一行数据），否则最好让 MySQL 自己处理 id 和 timestamp 字段。

现在表里有一些数据了，你可以用很多方法来选择这些数据。下面是几个 SELECT 语句的示例：

```
> SELECT * FROM pages WHERE id = 2;
```

这条语句告诉 MySQL，“从 pages 表中把 id 字段等于 2 的整行数据全挑选出来”。这个星号（*）是通配符，表示所有字段，这条语句会把满足条件（WHERE id = 2）的所有行的所有字段都显示出来。如果没有任何一行的 id 字段等于 2，就会返回一个空集。例如，下面这个不区分大小写的查询，会返回 title 字段里包含“test”的所有行（% 符号表示 MySQL 字符串通配符）的所有字段：

```
> SELECT * FROM pages WHERE title LIKE "%test%";
```

但是，如果你的表有很多字段，而你只想返回部分字段怎么办？你可以不用星号，而用下面的方式：

```
> SELECT id, title FROM pages WHERE content LIKE "%page content%";
```

这样就只会返回 content 字段包含“page content”的所有行的 id 和 title 两个字段了。

DELETE 语句的语法与 SELECT 语句类似：

```
> DELETE FROM pages WHERE id = 1;
```

由于数据库的数据删除后不能恢复，所以在执行 DELETE 语句之前，建议用 SELECT 确认一下要删除的数据（本例中，就是用 SELECT * FROM pages WHERE id = 1 查看），然后把 SELECT * 换成 DELETE 就可以了，这会是一个好习惯。很多程序员都有过 DELETE 误操作的伤心往事，还有一些人在慌乱中忘了在语句中放 WHERE，结果把所有客户数据都删除了。别让这种事发生在你身上！

在使用 UPDATE 语句时也要小心谨慎：

```
> UPDATE pages SET title="A new title",
content="Some new content" WHERE id=2;
```

结合本书的主题，后面我们就只用这些基本的 MySQL 语句，做一些简单的数据查询、创建和更新工作。如果你对这个强大的数据库工具的命令和技术感兴趣，推荐你去看 Paul DuBois 的 *MySQL Cookbook*。

6.3.3　与Python整合

Python 没有内置的 MySQL 支持工具。不过，有很多开源的库可以用来与 MySQL 做交互，Python 2.x 和 Python 3.x 版本都支持。其中最有名的一个库就是 PyMySQL。

写作本书的时候，PyMySQL 的版本是 0.6.7，可以用 pip 安装：

```
$ pip install PyMySQL
```

如果需要使用指定版本的 PyMySQL，可以下载源文件进行安装：

```
$ curl -L https://pypi.python.org/packages/source/P/PyMySQL/PyMySQL-0.6.7.tar.gz\
| tar xz
$ cd PyMySQL-PyMySQL-f953785/
$ python setup.py install
```

安装完成之后，你就可以使用 PyMySQL 包了。如果你的本地 MySQL 服务器处于运行状态，应该可以成功地执行下面的命令（记得把 root 账户密码加进数据库）：

```
import pymysql
conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
                       user='root', passwd=None, db='mysql')
cur = conn.cursor()
cur.execute('USE scraping')
cur.execute('SELECT * FROM pages WHERE id=1')
print(cur.fetchone())
cur.close()
conn.close()
```

这段程序有两个新对象类型：连接对象（conn）和光标对象（cur）。

连接 / 光标模式是数据库编程中常用的模式，不过刚刚接触数据库的时候，有些用户很难

区分这两种模式的不同。连接模式除了要连接数据库之外，还要发送数据库信息，处理回滚操作（当一个查询或一组查询被中断时，数据库需要回到初始状态，一般用事务控制手段实现状态回滚），创建新的光标对象，等等。

而一个连接可以有很多个光标。一个光标跟踪一种**状态**（state）信息，比如正在使用的是哪个数据库。如果你有多个数据库，且需要向所有数据库写内容，就需要多个光标来进行处理。光标还会包含最后一次查询执行的结果。通过调用光标函数，比如 `cur.fetchone()`，可以获取查询结果。

用完光标和连接之后，千万要记得把它们关闭。如果不关闭就会导致**连接泄漏**（connection leak），造成一种未关闭连接现象，即连接已经不再使用，但是数据库却不能关闭，因为数据库不确定你还要不要继续使用它。这种现象会一直耗费数据库的资源（我曾经写过也修复过很多连接泄漏 bug），所以用完数据库之后记得关闭连接！

刚开始的时候，你最想做的事情可能就是把抓取的结果保存到数据库里。让我们用前面维基百科爬虫的例子来演示一下如何实现数据存储。

在进行网页抓取时，处理 Unicode 字符串是很痛苦的事情。默认情况下，MySQL 也不支持 Unicode 字符处理。不过你可以开启这个功能（只是要记住，这么做会增加数据库的占用空间）。因为在维基百科上我们难免会遇到各种各样的字符，所以最好一开始就让你的数据库支持 Unicode：

```
ALTER DATABASE scraping CHARACTER SET = utf8mb4 COLLATE = utf8mb4_unicode_ci;
ALTER TABLE pages CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
ALTER TABLE pages CHANGE title title VARCHAR(200) CHARACTER SET utf8mb4 COLLATE
utf8mb4_unicode_ci;
ALTER TABLE pages CHANGE content content VARCHAR(10000) CHARACTER SET utf8mb4 CO
LLATE utf8mb4_unicode_ci;
```

这 4 行语句将数据库、数据表以及两个字段的默认编码都从 `utf8mb4`（严格说来也属于 Unicode，但是对大多数 Unicode 字符的支持都非常不好）转变成了 `utf8mb4_unicode_ci`。

你可以在 `title` 或 `content` 字段中插入一些德语变音符（umlauts）或汉语字符，如果没有错误，就表示转换成功了。

现在数据库已经准备好接收维基百科的各种信息了，你可以用下面的程序来存储数据：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import datetime
import random
import pymysql
import re

conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
                       user='root', passwd=None, db='mysql', charset='utf8')
```

```
cur = conn.cursor()
cur.execute("USE scraping")

random.seed(datetime.datetime.now())

def store(title, content):
    cur.execute('INSERT INTO pages (title, content) VALUES '
        '("%s", "%s")', (title, content))
    cur.connection.commit()

def getLinks(articleUrl):
    html = urlopen('http://en.wikipedia.org'+articleUrl)
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    content = bs.find('div', {'id':'mw-content-text'}).find('p')
        .get_text()
    store(title, content)
    return bs.find('div', {'id':'bodyContent'}).findAll('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))

links = getLinks('/wiki/Kevin_Bacon')
try:
    while len(links) > 0:
        newArticle = links[random.randint(0, len(links)-1)].attrs['href']
        print(newArticle)
        links = getLinks(newArticle)
finally:
    cur.close()
    conn.close()
```

这里有几点需要注意：首先，`charset='utf8'` 要添加到连接字符串里。这是让连接 `conn` 把所有发送到数据库的信息都当成 UTF-8 编码格式（当然，前提是数据库默认编码已经设置成 UTF-8）。

然后要注意的是 `store` 函数。它有两个参数：`title` 和 `content`，并把这两个参数添加到了一个 `INSERT` 语句中并用光标执行，然后用光标进行连接确认。这是一个让光标与连接操作分离的好例子；当光标里存储了与数据库和数据库上下文（context）相关的信息时，它需要通过连接的确认操作先将信息传进数据库，再将信息插入数据库。

最后要注意的是，`finally` 语句是在程序主循环的外面，代码的最底下。这样做可以保证，就算程序执行过程中发生中断或抛出异常（因为 Web 很复杂，所以你得随时准备遭遇异常），光标和连接都会在程序结束前立即关闭。无论你是在抓取网页，还是处理一个打开的数据库连接，用 `try...finally` 都是一个好主意。

虽然 PyMySQL 规模并不大，但是里面有很多非常实用的函数，本书无法一一介绍。具体请参考 PyMySQL 站点上的文档。

6.3.4 数据库技术与最佳实践

有些人的整个职业生涯都在学习、优化和创造数据库。我不是这类人，这本书也不是那类书。但是，和计算机科学的很多主题一样，有一些技巧你其实可以很快地学会，它们至少可以让你的数据库适用于大多数情况，而且运行速度足够快。

首先，总是给每个数据表都增加一个 `id` 字段。MySQL 里所有的表都至少有一个主键（就是 MySQL 用来排序的字段），因此 MySQL 知道怎么组织主键，通常数据库很难智能地选择主键。

究竟是用人造的 `id` 字段作为主键，还是用那些具有唯一性属性的字段作为主键，比如 `username` 字段，数据科学家和软件工程师已经争论了很多年，我更倾向于主动创建一个 `id` 字段。尤其是当你抓取并存储其他人的数据时，你并不知道哪些是唯一的哪些是非唯一的，至少我就遇到过这样的情况。

你应该用自增的 `id` 字段作为你的所有表格的主键。

其次，用智能索引。字典（指的是常用的工具书，不是指 Python 的字典对象）是按照字母顺序排列的单词表。这让你可以快速地找到一个单词，只要你知道这个单词是如何拼写的。你还可以想象一个将单词按照单词定义的字母顺序进行排列的字典。除非你在玩《危险边缘》（*Jeopardy*）这样奇怪的智力游戏，给出定义，让你猜单词，否则这样的字典就没什么用了。但是在数据库查询中，这种按照字段含义进行排序的情况时有发生。比如，你的数据库里可能有一个字段经常要查询：

```
>SELECT * FROM dictionary WHERE definition="A small furry animal that says meow";
+------+-------+-------------------------------------+
|  id  | word  | definition                          |
+------+-------+-------------------------------------+
| 200  |  cat  | A small furry animal that says meow |
+------+-------+-------------------------------------+
1 row in set (0.00 sec)
```

你可能想给这个表的 `definition` 字段添加一个索引（除了 `id` 字段可能已经存在的索引之外），让这个字段的查询变得更快。但是，增加索引需要占用更多的空间，而且插入新行的时候也需要更多的处理时间。尤其是当处理大量的数据时，你应该仔细权衡你的索引和你需要多少索引。为了让这个“定义”索引简单点儿，你可以让 MySQL 只检索字段值的一部分字符。比如下面的命令创建了一个查询 `definition` 字段前 16 个字符的智能索引：

```
CREATE INDEX definition ON dictionary (id, definition(16));
```

在根据完整定义搜索单词时，这个索引会使查询速度快很多（尤其是前 16 个字符彼此有很大不同时），而且不需要占用过多的空间和处理时间。

关于数据查询时间和数据库大小（数据库工程中一个基本的平衡做法），一个常见的错误就是在数据库中存储大量重复数据，尤其是在对大量自然语言数据进行网页抓取时。举个例子，假设你想统计网站上突然出现的一些词组的频率。这些词组也许可以从一个现成的列表里获得，也许可以通过文本分析算法自动提取。最终你可能会把词组存储成如下形式：

```
+--------+--------------+------+-----+---------+----------------+
| Field  | Type         | Null | Key | Default | Extra          |
+--------+--------------+------+-----+---------+----------------+
| id     | int(11)      | NO   | PRI | NULL    | auto_increment |
| url    | varchar(200) | YES  |     | NULL    |                |
| phrase | varchar(200) | YES  |     | NULL    |                |
+--------+--------------+------+-----+---------+----------------+
```

每当你发现一个词组，就在数据库中增加一行，同时把 URL 记录下来。但是，如果把这些数据分成 3 个表，数据库占用的空间就会大大减少。

```
>DESCRIBE phrases
+--------+--------------+------+-----+---------+----------------+
| Field  | Type         | Null | Key | Default | Extra          |
+--------+--------------+------+-----+---------+----------------+
| id     | int(11)      | NO   | PRI | NULL    | auto_increment |
| phrase | varchar(200) | YES  |     | NULL    |                |
+--------+--------------+------+-----+---------+----------------+

>DESCRIBE urls
+--------+--------------+------+-----+---------+----------------+
| Field  | Type         | Null | Key | Default | Extra          |
+--------+--------------+------+-----+---------+----------------+
| id     | int(11)      | NO   | PRI | NULL    | auto_increment |
| url    | varchar(200) | YES  |     | NULL    |                |
+--------+--------------+------+-----+---------+----------------+

>DESCRIBE foundInstances
+-------------+---------+------+-----+---------+----------------+
| Field       | Type    | Null | Key | Default | Extra          |
+-------------+---------+------+-----+---------+----------------+
| id          | int(11) | NO   | PRI | NULL    | auto_increment |
| urlId       | int(11) | YES  |     | NULL    |                |
| phraseId    | int(11) | YES  |     | NULL    |                |
| occurrences | int(11) | YES  |     | NULL    |                |
+-------------+---------+------+-----+---------+----------------+
```

虽然表定义的结构变复杂了，但是大部分字段都是 `id` 字段。它们是整数，不会占用很多空间。另外，每个 URL 和词组都只会存储一次。

除非你安装了第三方包或保存详细的数据库日志，否则你无法掌握数据库里数据增加、更新或删除的具体时间。因此，取决于数据的可用空间、变更的频率以及确定变更时间的重要性，你可以考虑在创建、更新或删除数据时加一个时间戳。

6.3.5 MySQL里的“六度空间游戏”

在第 3 章，我们介绍过“维基百科六度分隔”问题，其目标是通过一些词条链接寻找两个词条间的联系（即找出一条链接路径，只要点击链接就可以从一个维基词条到达另一个维基词条）。为了解决这个问题，我们不仅需要建立网络爬虫抓取网页（之前我们已经做过），还要把抓取的信息以某种形式存储起来，以便后续进行数据分析。

前面介绍过的自增的 `id` 字段、时间戳以及多份数据表这里都要用到。为了确定最合理的信息存储方式，你需要进行抽象思考。一个链接可以轻易地把页面 A 连接到页面 B。同样也可以轻易地把页面 B 连接到页面 A，不过这可能是另一个链接。我们可以这样识别一个链接，即“页面 A 存在一个链接，可以连接到页面 B。也就是 `INSERT INTO links (fromPageId, toPageId) VALUES (A, B);`（其中，A 和 B 分别表示两个页面的 ID 号）”。

因此需要设计一个带有两张数据表的数据库来分别存储页面和链接，两张表都带有创建时间和唯一的 ID 号，代码如下所示：

```
CREATE TABLE `wikipedia`.`pages` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `url` VARCHAR(255) NOT NULL,
  `created` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`));

CREATE TABLE `wikipedia`.`links` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `fromPageId` INT NULL,
  `toPageId` INT NULL,
  `created` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`));
```

注意，这里和前面打印页面标题的爬虫不同，我没有在页面数据表里使用页面标题字段。为什么呢？其实是因为页面标题得在你进入页面后才能抓到。如果我们想创建一个高效的爬虫来填充这些数据表，那么只存储页面的链接就可以保存词条页面了，甚至不需要访问词条页面。

当然，并不是所有网站都具有这个特点，但是维基百科的词条链接和对应的页面标题是可以通过简单的操作进行转换的。例如，http://en.wikipedia.org/wiki/Monty_Python 表明了页面标题是“Monty Python”。

下面的代码会把“贝肯数”（一个页面与凯文·贝肯词条页面之间的链接数）不超过 6 的维基百科页面存储起来：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import pymysql
from random import shuffle
```

```
conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
                       user='root', passwd=None, db='mysql', charset='utf8')
cur = conn.cursor()
cur.execute('USE wikipedia')

def insertPageIfNotExists(url):
    cur.execute('SELECT * FROM pages WHERE url = %s', (url))
    if cur.rowcount == 0:
        cur.execute('INSERT INTO pages (url) VALUES (%s)', (url))
        conn.commit()
        return cur.lastrowid
    else:
        return cur.fetchone()[0]

def loadPages():
    cur.execute('SELECT * FROM pages')
    pages = [row[1] for row in cur.fetchall()]
    return pages

def insertLink(fromPageId, toPageId):
    cur.execute('SELECT * FROM links WHERE fromPageId = %s '
        'AND toPageId = %s', (int(fromPageId), int(toPageId)))
    if cur.rowcount == 0:
        cur.execute('INSERT INTO links (fromPageId, toPageId) VALUES (%s, %s)',
                    (int(fromPageId), int(toPageId)))
        conn.commit()


def getLinks(pageUrl, recursionLevel, pages):
    if recursionLevel > 4:
        return

    pageId = insertPageIfNotExists(pageUrl)
    html = urlopen('http://en.wikipedia.org{}'.format(pageUrl))
    bs = BeautifulSoup(html, 'html.parser')
    links = bs.findAll('a', href=re.compile('^(/wiki/)((?!:).)*$'))
    links = [link.attrs['href'] for link in links]

    for link in links:
        insertLink(pageId, insertPageIfNotExists(link))
        if link not in pages:
            # 遇到一个新页面，加入集合并搜索里面的词条链接
            pages.append(link)
            getLinks(link, recursionLevel+1, pages)

getLinks('/wiki/Kevin_Bacon', 0, loadPages())
cur.close()
conn.close()
```

这里有 3 个函数使用 PyMySQL 与数据库进行了交互。

`insertPageIfNotExists`

正如其名称所示，当页面不存在时，该函数会插入一个新的页面记录。该页面以及其他已经抓取的页面作为列表存储在 `pages` 变量中，以确保页面记录不会重复。它也提供了一个供查询的 `pageId` 数，以创建新的链接。

`insertLink`

该函数在数据库中创建一个新的链接。如果该链接已经存在，则不会创建。如果同一个页面中存在两个或者多个相同的链接，我们会将其当作同一个链接，表示同样的关系，并且应该被当作一条记录。这样，如果程序对同一页面运行多遍，也有助于维护数据库的一致性。

`loadPages`

该函数将当前所有页面从数据库加载到一个列表中，这样可以确定新的页面是否被访问过。在程序运行时也会收集页面，因此如果从一个空的数据库开始，爬虫仅仅运行一遍，那么理论上说 `loadPages` 是不需要的。实际上，这会导致问题。原因是网络可能会中断，或者你希望在不同的时间段抓取各链接，因此让爬虫可以重新自我加载非常重要。

你应该注意的是使用 `loadPages` 可能导致的潜在问题，以及为了确定页面是否被访问过而生成的页面列表：当每个页面被加载时，页面上所有的链接会被立刻存储为页面，即使这些页面从未被访问过——仅仅是这些链接被发现了。如果爬虫被停止然后重启，所有这些“被发现但是未访问的”页面将永远不会被访问，而来自这些页面的链接也不会被记录下来。该问题可以通过在每个页面记录中加入一个布尔变量 `visited` 来解决，并且仅当页面被加载以及其自身的外链被记录，才将该变量的值设置为 `True`。

对于我们的目标来说，这个解决方案是可行的。如果你能保证足够长的运行时间（或者是单次运行时间），那么完整的链接集合（用于实验的较大数据集）并不是必需的，也就意味着 `visited` 变量不是必需的。

关于该问题以及从 Kevin Bacon（https://en.wikipedia.org/wiki/Kevin_Bacon）到 Eric Idle（https://en.wikipedia.org/wiki/Eric_Idle）的最终解决方案，请查看 9.2 节中关于有向图问题的求解。

6.4 Email

与网页通过 HTTP 协议传输一样，邮件是通过 SMTP（Simple Mail Transfer Protocol，简单邮件传输协议）传输的。而且，与你用 Web 服务器的客户端（浏览器）通过 HTTP 协议传输网页一样，服务器使用各种 Email 客户端收发邮件，比如 Sendmail、Postfix 和 Mailman 等。

虽然用 Python 发邮件很容易，但是需要你连接一台运行 SMTP 协议的服务器。在服务器或本地机器上设置 SMTP 客户端有点儿复杂，也超出了本书的范围，但是有很多资料可以帮你解决问题，如果你用的是 Linux 或 macOS 系统，参考资料会更丰富。

下面的代码运行的前提是你的电脑正在运行一个 SMTP 客户端。（如果要调整代码用于远程 SMTP 客户端，请把 `localhost` 改成远程服务器的地址。）

用 Python 发一封邮件只要 9 行代码：

```
import smtplib
from email.mime.text import MIMEText

msg = MIMEText('The body of the email is here')

msg['Subject'] = 'An Email Alert'
msg['From'] = 'ryan@pythonscraping.com'
msg['To'] = 'webmaster@pythonscraping.com'

s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

Python 有两个重要的包可以发送邮件：`smtplib` 和 `email`。

Python 的 `email` 模块里包含了许多实用的邮件格式设置函数，可以用来创建邮件“包裹”。示例中使用的 `MIMEText` 对象为底层的 MIME（Multipurpose Internet Mail Extensions，多用途互联网邮件扩展）协议传输创建了一封空邮件，最后通过高层的 SMTP 协议发送出去。`MIMEText` 对象 `msg` 包括收发邮件地址、邮件正文和主题，Python 通过它就可以创建一封格式正确的邮件。

`smtplib` 模块用来设置服务器连接的相关信息。就像 MySQL 服务器的连接一样，这个连接必须在用完之后及时关闭，以避免同时创建太多连接而浪费资源。

把这个简单的邮件程序封装成函数后，可以更方便地扩展和使用：

```
import smtplib
from email.mime.text import MIMEText
from bs4 import BeautifulSoup
from urllib.request import urlopen
import time

def sendMail(subject, body):
    msg = MIMEText(body)
    msg['Subject'] = subject
    msg['From'] ='christmas_alerts@pythonscraping.com'
    msg['To'] = 'ryan@pythonscraping.com'

    s = smtplib.SMTP('localhost')
```

```
        s.send_message(msg)
        s.quit()

bs = BeautifulSoup(urlopen('https://isitchristmas.com/'), 'html.parser')
while(bs.find('a', {'id':'answer'}).attrs['title'] == 'NO'):
    print('It is not Christmas yet.')
    time.sleep(3600)
    bs = BeautifulSoup(urlopen('https://isitchristmas.com/'), 'html.parser')

sendMail('It\'s Christmas!',
         'According to https://isitchristmas.com, it is Christmas!')
```

这个程序每小时检查一次 https://isitchristmas.com/ 网站（根据日期判断当天是不是圣诞节）。如果页面上的信息不是“NO”[4]，就会给你发一封邮件，告诉你圣诞节到了。

虽然这个程序看起来并没有墙上的挂历有用，但是稍作修改就可以做很多有用的事情。它可以发送网站访问失败、应用测试失败的异常情况，也可以在 Amazon 网站上出现了一款卖到断货的畅销品时通知你——这些都是挂历做不到的事情。

注 4：中国用户在网站页面上看到的“NO”在源代码里是“不是”。——译者注

第二部分

高级网页抓取

你已经掌握了网页抓取的一些基础知识，现在让我们进入更有趣的第二部分。到目前为止，我们创建的网络爬虫还不是特别给力。如果 Web 服务器不能立即提供样式规范的信息，爬虫就不能正确地抓取数据。如果爬虫只能抓取那些显而易见的信息，不经过处理就简单地存储起来，那么迟早要被登录表单、网站交互以及 JavaScript 困住手脚。总之，目前爬虫还没有足够的实力去抓取各种数据，只能处理那些愿意被抓取的信息。

这部分内容就是要帮你分析原始数据，获取隐藏在数据背后的故事——网站的真实故事其实都隐藏在 JavaScript、登录表单和网站反抓取措施的背后。通过学习这部分内容，你将掌握如何用网络爬虫测试网站，自动化处理，以及通过更多的方式接入网络。最后你将学到一些数据抓取工具，它们能够帮助你深入互联网的每个角落，收集和操作几乎所有类型的数据。

第7章

读取文档

有种观点认为，互联网基本上就是那些符合新式 Web 2.0 潮流，并且经过多媒体内容点缀的 HTML 网站构成的集合，这些内容在网页抓取时几乎都是可以忽略的。但是，这种观点忽略了互联网最基本的特征：作为不同类型文件的传输媒介。

虽然互联网在 20 世纪 60 年代末期就已经以不同的形式出现，但是 HTML 直到 1992 年才问世。在此之前，互联网基本上就是用来收发邮件和传输文件，那时还没有“网页”的概念。换言之，互联网并不是一个 HTML 页面的集合。它是一个由多种类型的文档构成的集合，而 HTML 文件经常被用作展示文档的一个框架。如果不能读取各种类型的文档，包括纯文本、PDF、图像、视频、邮件等，我们将会遗漏很大一部分可用数据。

本章重点介绍文档处理的相关内容，包括把文档下载到本地文件夹里，以及读取文档并提取数据。还会介绍文档的不同编码类型，让程序可以读取非英文的 HTML 页面。

7.1　文档编码

文档编码告诉程序——无论是计算机的操作系统还是你自己的 Python 代码——如何读取文档。文档编码的方式通常可以根据文件的扩展名进行判断，虽然文件扩展名并不是由编码决定的，而是由开发者决定的。例如，如果我把 myImage.jpg 另存为 myImage.txt，不会出现任何问题，但当我用文本编辑器打开它的时候就有问题了。好在这种情况很少见，要正确地读取一个文档，通常只需知道它的扩展名。

从根本上说，所有文档都是由 0 和 1 编码而成的。除此之外，编码算法会定义“每个字符多少位”或“每个像素的颜色值用多少位”（图像文件里）之类的事情。另外，你可能有

一层数据压缩算法或体积缩减算法，比如 PNG 图像编码格式（一种无损压缩的位图图形格式）。

虽然第一次处理非 HTML 格式的文件时会觉得很没底，但是只要安装了合适的库，Python 就可以帮你处理任意类型的文档。纯文本文件、视频文件和图像文件的唯一区别，就是它们的 0 和 1 面向用户的转换方式不同。本章会介绍几种常用的文档格式：纯文本、CSV、PDF 和 Word 文档。

这些文档格式基本上都是用来存储文字的。如果你需要关于图像处理的信息，那么我推荐先通读这一章，掌握处理和存储不同文件类型的方法，再阅读第 13 章关于图像处理的内容！

7.2　纯文本

虽然把文件存储为在线的纯文本格式并不常见，但是简易网站或者旧式网站经常有大量的纯文本文件。例如，互联网工程任务组（Internet Engineering Task Force，IETF）网站就存储了其发表过的所有文档，包含 HTML、PDF 和纯文本格式（例如 https://www.ietf.org/rfc/rfc1149.txt）。大多数浏览器都可以很好地显示纯文本文件，抓取它们也不会遇到什么问题。

对于大多数简单的纯文本文件，比如 http://www.pythonscraping.com/pages/warandpeace/chapter1.txt 这个练习文件，可以用下面的方法读取：

```
from urllib.request import urlopen
textPage = urlopen('http://www.pythonscraping.com/'\
    'pages/warandpeace/chapter1.txt')
print(textPage.read())
```

通常，当用 `urlopen` 获取了网页之后，我们会把它转变成 `BeautifulSoup` 对象，以方便后面对 HTML 进行解析。在这里，我们可以直接读取页面内容。虽然完全可以把它转变成 `BeautifulSoup` 对象，但这样做其实适得其反——这个页面不是 HTML，所以 BeautifulSoup 库就没用了。一旦纯文本文件被读成字符串，你就只能用普通 Python 字符串的方法分析它了。当然，这么做有个缺点，就是你不能对字符串使用 HTML 标签，去定位那些你真正需要的文字，避开那些你不需要的文字。如果现在你想从纯文本文件中抽取某些信息，还是有些难度的。

文本编码和全球互联网

前面说过，如果想正确地读取一个文件，只需知道它的扩展名就可以了。不过非常奇怪的是，这条规则不能应用到最基本的文档格式：.txt 文件。

大多数时候，用前面介绍的方法读取纯文本文件都没有问题。但是，互联网上的文本文件

会比较复杂。下面介绍一些英文和非英文编码的基础知识，包括 ASCII、Unicode 和 ISO 编码，以及对应的处理方法。

1. 文本编码类型简介

ASCII 是在 20 世纪 60 年代首次发明的一套编码系统，当时比特还非常昂贵，并且也没必要编码除拉丁字母和一些标点符号外的任何东西。因此，对于编码 128 个大写字母、小写字母和标点符号来说，7 位就够了。还有 33 个非打印字符，其中有些被使用，有些被替换，有些随着这些年技术的发展被废弃了。这样看来空间还是很多的，不是吗？

每一位程序员都知道，7 是一个奇怪的数字。它并不是 2 的幂，但是也很接近。20 世纪 60 年代，究竟是应该增加一位以获得一个漂亮的二进制数（用 8 位），还是让文件占用更少的存储空间（用 7 位），计算机科学家们对此争论不休。最终，7 位编码胜利了。但是，在新式的计算方式中，每个 7 位码的前面都补充（pad）了一个“0”[1]，留给我们两个最坏的结果是，文件大了 14%（编码由 7 位变成 8 位，体积增加了 14%），并且由于只有 128 个字符，缺乏灵活性。

20 世纪 90 年代初，人们认识到世界上除了英语还存在其他很多语言，如果计算机也可以显示这些语言就太好了。一个叫 Unicode 联盟（The Unicode Consortium）的非营利组织尝试对地球上所有用于书写的字符进行统一编码。其目标包括拉丁字母、斯拉夫字母（кириллица）、中国象形文字（象形）、数学和逻辑符号（∑、≥），甚至表情符号和其他符号，如生化危机标记（☣）和和平符号（☮）等。

编码的结果就是你可能已熟知的 UTF-8（Universal Character Set — Transformation Format 8 bit，统一字符集—转换格式 8 位）。“8 位”指的不是每个字符的大小，而是显示一个字符所需要的最小位数。

UTF-8 字符的实际大小是非常灵活的，范围从 1 字节到 4 字节，具体取决于它们在可能的字符列表中的位置（常用字符用更少的字节编码，相对罕见的字符则需要更多字节）。

那么 UTF-8 的灵活性是如何实现的呢？ ASCII 码的 7 位编码以及毫无用处的开头补零乍看起来像是一个设计错误，但实际上却是 UTF-8 的一大优势。因为 ASCII 非常受欢迎，所以 Unicode 决定利用开头补零，让所有以“0”开头的字节表示这个字符只用 1 个字节，从而使得 ASCII 和 UTF-8 的编码机制完全一样。因此，下面的字符在 ASCII 和 UTF-8 两种编码方式中都是有效的：

```
01000001 - A
01000010 - B
01000011 - C
```

注 1：padding（填充）位在稍后介绍 ISO 编码标准时还会提到。

而下面的字符只在 UTF-8 编码里有效，如果文档用 ASCII 编码，它们就会被看成“无法打印”：

```
11000011 10000000 - À
11000011 10011111 - ß
11000011 10100111 - ç
```

除了 UTF-8，还有其他 UTF 标准，比如 UTF-16、UTF-24 和 UTF-32，不过很少用这些编码标准对文件进行编码，仅限于特殊情况，而这超出了本书范围。

尽管 ASCII 最初的“设计错误”给 UTF-8 带来了很大的便利，但是其缺陷也并未完全消失。每个字符前 8 位的信息仍然只能编码 128 个字符，而不是完整的 256 个字符。在需要多个字节的 UTF-8 字符中，额外补齐的位并不是用于字符的编码，而是用于校验位以防止产生歧义。四字节字符的 32（8 × 4）位中，只有 21 位用于为总共 2 097 152 个可能的字符（其中 1 114 112 个字符已分配）进行编码。

当然，所有通用语言编码标准的问题就是任何一种非英文语言文档的体积都比 ASCII 编码的体积大。虽然你的语言可能只由大约 100 个字符构成，但是你还是得用 16 位表示每个字符，而不只是 8 位，就像英文的 ASCII 编码。这会让采用 UTF-8 编码的非英文的纯文本文档的体积差不多达到英文文档的两倍，至少对那些不用拉丁字符集的语言来说是如此。

ISO 标准解决这个问题的办法是为每种语言创建一种编码。和 Unicode 一样，它使用了与 ASCII 相同的编码，但是在每个字符的开头用 0 作“填充位”，这样就可以为所有语言创建 128 个特殊字符。这种做法对那些依赖拉丁文字母的欧洲语言（编码还是按照 0–127 一一对应）非常合适，只不过需要增加一些特殊字符。这使得 ISO-8859-1（为拉丁文字母设计的）标准里有了分数符号（如 ½）和版权标记（©）。

还有一些 ISO 字符集，像 ISO-8859-9（土耳其语）、ISO-8859-2（德语等语言）、ISO-8859-15（法语等语言）也是用类似的规律做出来的。

虽然这些年 ISO 编码标准的使用率一直在下降，但是目前仍有约 9% 的网站使用 ISO 编码[2]，所以在抓取网站之前有必要了解并检查是否使用了这种编码方法。

2. 编码进行时

在上一节里，我们用采取默认设置的 `urlopen` 读取了网上的纯文本文档。这么做对大多数英文文本来说没有任何问题。但是，如果你遇到的是俄语、阿拉伯语，或者是像“résumé”这样的单词，就可能会出问题。

看看下面的代码：

注 2：数据源自 http://w3techs.com/technologies/history_overview/character_encoding，通过网络爬虫收集。

```
from urllib.request import urlopen
textPage = urlopen('http://www.pythonscraping.com/'\
    'pages/warandpeace/chapter1-ru.txt')
print(textPage.read())
```

这段代码会把《战争与和平》原著（托尔斯泰用俄语和法语写的）的第 1 章打印到屏幕上。打印结果一开头是这样：

```
b"\xd0\xa7\xd0\x90\xd0\xa1\xd0\xa2\xd0\xac \xd0\x9f\xd0\x95\xd0\xa0\xd0\x92\xd0\
x90\xd0\xaf\n\nI\n\n\xe2\x80\x94 Eh bien, mon prince.
```

另外，在大多数浏览器里访问该页面会呈现乱码（参见图 7-1）。

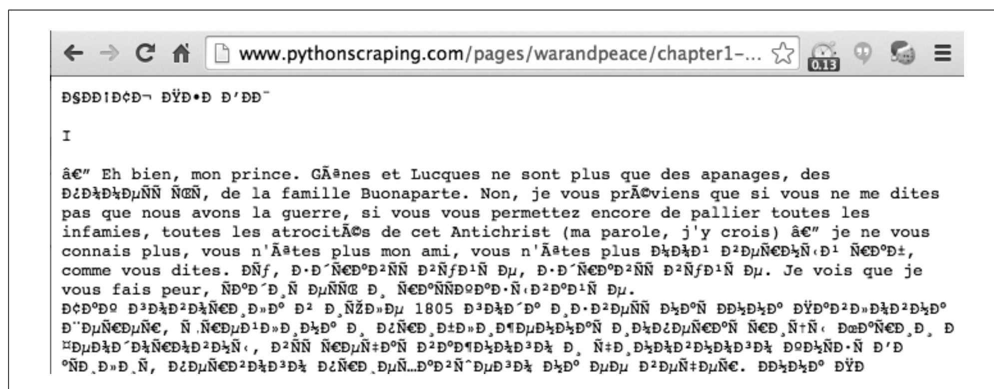


图 7-1：法语和斯拉夫语文本用 ISO-8859-1（许多浏览器默认的文本编码格式）编码的效果

就算让以俄语为母语的人来看，这些乱码也难以辨认。问题在于，Python 试图把文本读成 ASCII 编码格式，而浏览器试图把文本读成 ISO-8859-1 编码格式。其实都不对，应该是 UTF-8 编码格式。

我们可以把字符串显示转换成 UTF-8 格式，这样就可以正确显示斯拉夫文字了：

```
from urllib.request import urlopen

textPage = urlopen('http://www.pythonscraping.com/'\
    'pages/warandpeace/chapter1-ru.txt')
print(str(textPage.read(), 'utf-8'))
```

用 BeautifulSoup 和 Python 3.x 对文档进行 UTF-8 编码，如下所示：

```
html = urlopen('http://en.wikipedia.org/wiki/Python_(programming_language)')
bs = BeautifulSoup(html, 'html.parser')
content = bs.find('div', {'id':'mw-content-text'}).get_text()
content = bytes(content, 'UTF-8')
content = content.decode('UTF-8')
```

Python 3.x 默认将所有字符编码成 UTF-8。你可能打算以后用网络爬虫的时候全部采用

UTF-8 编码读取内容，毕竟 UTF-8 也可以完美地处理 ASCII 字符和非英语语言。但是，要记住还有 9% 的网站使用 ISO 编码格式，所以你不能完全避免该问题。

不幸的是，在处理纯文本文档时，无法具体确定文档的编码。有一些库可以检查文档的编码，或是对文档编码进行估计（用一些逻辑来判断“Ñ€Ð°ÑÑÐºÐ°Ð·Ñ”很可能不是单词），不过效果并不是很好。

幸运的是，在处理 HTML 页面的时候，编码格式通常会包含在网站 `<head>` 部分的标签中。大多数网站，尤其是英文网站，都会带这样的标签：

```
<meta charset="utf-8" />
```

而 ECMA（European Computer Manufacturers Association，欧洲计算机制造商协会）网站的标签是这样的[3]：

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
```

如果你要做很多网页抓取工作，尤其是面对国际网站时，建议你先看看 `meta` 标签的内容，用网站推荐的编码方式读取页面内容。

7.3 CSV

进行网页抓取的时候，你可能会遇到 CSV 文件，也可能有同事希望将数据保存为 CSV 格式。Python 有一个超赞的标准库（https://docs.python.org/3.4/library/csv.html）可以读写 CSV 文件。虽然这个库可以处理各种 CSV 文件，但是本节重点介绍标准 CSV 格式。如果你在处理 CSV 时有特殊需求，请查看文档！

读取CSV文件

Python 的 `csv` 库主要是面向本地文件，就是说你的 CSV 文件得存储在你的电脑上。而进行网页抓取的时候，很多文件都是在线的。不过有一些方法可以解决这个问题：

- 手动把 CSV 文件下载到本机，然后用 Python 定位文件位置；
- 写 Python 程序下载文件，读取文件，之后（可以）把源文件删除；
- 从网上直接把文件读成一个字符串，然后转换成一个 `StringIO` 对象，使它具有文件的属性。

虽然前两种方法也可行，但是既然你可以轻易地把 CSV 文件保存在内存里，就不要再下载到本地占用硬盘空间了。直接把文件读成字符串，然后封装成 `StringIO` 对象，让 Python 把它当作文件来处理，就不需要先保存文件了。下面的程序就是从网上获取一个 CSV 文

注 3：ECMA 是 ISO 编码标准的主要贡献者之一，所以它的网站用 ISO 编码一点儿也不奇怪。

件（这里是 http://pythonscraping.com/files/MontyPythonAlbums.csv 里的 Monty Python 乐团的专辑列表），然后把每一行都打印到命令行里：

```
from urllib.request import urlopen
from io import StringIO
import csv

data = urlopen('http://pythonscraping.com/files/MontyPythonAlbums.csv')
              .read().decode('ascii', 'ignore')
dataFile = StringIO(data)
csvReader = csv.reader(dataFile)

for row in csvReader:
    print(row)
```

输出如下所示：

```
['Name', 'Year']
["Monty Python's Flying Circus", '1970']
['Another Monty Python Record', '1971']
["Monty Python's Previous Record", '1972']
...
```

从代码中你会发现 `csv.reader` 返回的 `csvReader` 对象是可迭代的，而且由 Python 的列表对象构成。因此，`csvReader` 对象的每一行可以用下面的方式获取：

```
for row in csvReader:
    print('The album "'+row[0]+'" was released in '+str(row[1]))
```

输出结果是：

```
The album "Name" was released in Year
The album "Monty Python's Flying Circus" was released in 1970
The album "Another Monty Python Record" was released in 1971
The album "Monty Python's Previous Record" was released in 1972
...
```

注意看第一行的内容，`The album "Name" was released in Year`。虽然写示例代码的时候，这行内容是否显示都无所谓，但是工作中你肯定不希望将这行信息保留在数据里。有些程序员可能会简单地跳过 `csvReader` 对象的第一行，或者写一个简单的条件把第一行处理掉。不过，还有一个函数可以很好地处理这个问题，那就是 `csv.DictReader`：

```
from urllib.request import urlopen
from io import StringIO
import csv

data = urlopen('http://pythonscraping.com/files/MontyPythonAlbums.csv')
              .read().decode('ascii', 'ignore')
dataFile = StringIO(data)
dictReader = csv.DictReader(dataFile)
```

```
print(dictReader.fieldnames)

for row in dictReader:
    print(row)
```

csv.DictReader 会把 CSV 文件的每一行转换成 Python 的字典对象返回，而不是列表对象，并把字段名称保存在变量 dictReader.fieldnames 里，作为字典对象的键：

```
['Name', 'Year']
{'Name': 'Monty Python's Flying Circus', 'Year': '1970'}
{'Name': 'Another Monty Python Record', 'Year': '1971'}
{'Name': 'Monty Python's Previous Record', 'Year': '1972'}
```

当然，与 csvReader 相比，创建、处理和打印这些 DictReader 对象要多花点时间，但是考虑到它的便利性和实用性，还是值得的。还要注意的是，在进行网页抓取的时候，无论写什么样的爬虫程序，从外部服务器请求和检索网站数据的时间消耗几乎都是不可避免的限制因素，因此担心两种技术中哪种可能会增加几微秒运行时间，其实没有什么实际意义。

7.4 PDF

作为一名 Linux 用户，我能理解电脑上没有微软软件却收到了一个 .docx 文件的痛苦，还有费半天劲儿找一种能够读取苹果系统媒体文件的解码器。从某种意义上说，Adobe 在 1993 年发明 PDF（Portable Document Format，便携式文档格式）是一种技术革命。PDF 让用户可以在不同的系统上用同样的方式查看图片和文本文档。

虽然把 PDF 存储在 Web 上已经有点儿过时了（你已经可以把内容写成 HTML 了，为什么还要用这种静态、加载速度超慢的格式存储内容呢？），但是 PDF 仍然无处不在，尤其是在处理商务报表和表单的时候。

2009 年，一个叫 Nick Innes 的英国人上了新闻，他根据英联邦的《信息自由法案》，要求英国白金汉郡议会公开学生的考试成绩。在几次请求遭到拒绝之后，他最终获得了所寻找的信息——184 份 PDF 文件。

虽然 Innes 努力坚持，并且最后得到了一个格式更好的数据库，但是如果他事先了解网络爬虫，再用 Python 众多 PDF 解析模块中的任意一个来直接处理这些 PDF 文件，那么他一定可以在法庭上节省很多时间。

不过目前很多 PDF 解析库都是用 Python 2.x 版本建立的，还没有迁移到 Python 3.x 版本。但是，因为 PDF 比较简单，而且是开源的文档格式，所以很多给力的 Python 库都可以读取 PDF 文件，而且支持 Python 3.x 版本。

PDFMiner3K 就是一个非常好用的库[4]。它非常灵活，可以通过命令行使用，也可以整合到代码中。它还可以处理不同的语言编码——对网页抓取而言非常方便。

你可以使用 pip 进行安装，也可以下载这个 Python 模块（https://pypi.python.org/pypi/pdfminer3k），然后解压并用下面的命令安装：

```
$ python setup.py install
```

文档位于源文件解压文件夹的 /pdfminer3k-1.3.0/docs/index.html 里，这个文档更多是在介绍命令行接口，而不是 Python 代码整合。

下面的例子可以把任意 PDF 读成字符串，然后用 `StringIO` 转换成文件对象：

```
from urllib.request import urlopen
from pdfminer.pdfinterp import PDFResourceManager, process_pdf
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from io import StringIO
from io import open

def readPDF(pdfFile):
    rsrcmgr = PDFResourceManager()
    retstr = StringIO()
    laparams = LAParams()
    device = TextConverter(rsrcmgr, retstr, laparams=laparams)

    process_pdf(rsrcmgr, device, pdfFile)
    device.close()

    content = retstr.getvalue()
    retstr.close()
    return content

pdfFile = urlopen('http://pythonscraping.com/'
    'pages/warandpeace/chapter1.pdf')
outputString = readPDF(pdfFile)
print(outputString)
pdfFile.close()
```

上面程序的文本输出如下：

```
CHAPTER I

"Well, Prince, so Genoa and Lucca are now just family estates of
the Buonapartes. But I warn you, if you don't tell me that this
means war, if you still try to defend the infamies and horrors
perpetrated by that Antichrist- I really believe he is Antichrist- I will
```

`readPDF` 函数的好处是，如果你的 PDF 文件在电脑里，你就可以直接把 `urlopen` 返回的对

注 4：它是 PDFMiner 的 Python 3.x 移植版。——译者注

象 `pdfFile` 替换成普通的 `open()` 文件对象：

```
pdfFile = open('../pages/warandpeace/chapter1.pdf', 'rb')
```

输出结果可能不是很完美，尤其是当 PDF 里有图片、各种各样的文本格式，或者带有表格和数据图的时候。但是，对大多数只包含纯文本内容的 PDF 而言，其输出结果与纯文本格式基本没什么区别。

7.5　微软Word和.docx

冒着冒犯微软朋友的风险说句话：我不喜欢微软的 Word 软件。并不是因为它是一款烂软件，而且因为它的用户误用了它。Word 的"特异功能"就是把那些应该写成简单的 TXT 或 PDF 格式的文件，变成了既大又慢且难以打开的"怪物"，而且它们经常在系统切换和版本切换中出现格式不兼容，并且因为某些原因在文件内容已经定稿后仍处于可编辑状态。

Word 文件被设计用于内容创建，而不是内容共享。不过它们在一些网站上很流行，包含重要的文档、信息，甚至是图表和多媒体，总之就是能够并且应该用 HTML 创建的一切。

大约在 2008 年以前，微软 Office 产品采用 .doc 文件格式。这种二进制文件格式很难读取，而且其他文字处理软件对它的支持也不好。为了跟上时代，让自己的软件能够符合主流软件的标准，微软决定使用基于 Office Open XML 的标准，此后新版 Word 文件才与其他文字处理软件兼容，这个格式就是 .docx。

不过，Python 对这种 Google Docs、Open Office 和 Microsoft Office 都在使用的 .docx 格式的支持还不够好。虽然有一个 python-docx 库，但是只支持创建新文档和读取一些基本的文件数据，如文件大小和文件标题，不支持正文读取。如果想读取 Microsoft Office 文件的正文内容，需要自己动手找方法。

第一步是从文件中读取 XML：

```
from zipfile import ZipFile
from urllib.request import urlopen
from io import BytesIO

wordFile = urlopen('http://pythonscraping.com/pages/AWordDocument.docx').read()
wordFile = BytesIO(wordFile)
document = ZipFile(wordFile)
xml_content = document.read('word/document.xml')
print(xml_content.decode('utf-8'))
```

这段代码把一个远程 Word 文档读成一个二进制文件对象（`BytesIO` 与本章前面用的 `StringIO` 类似），再用 Python 的标准库 `zipfile` 解压（为了节省空间，所有的 .docx 文件都进行过压缩），然后读取这个解压文件，就变成 XML 了。

这个 Word 文档在 http://pythonscraping.com/pages/AWordDocument.docx，内容如图 7-2 所示。

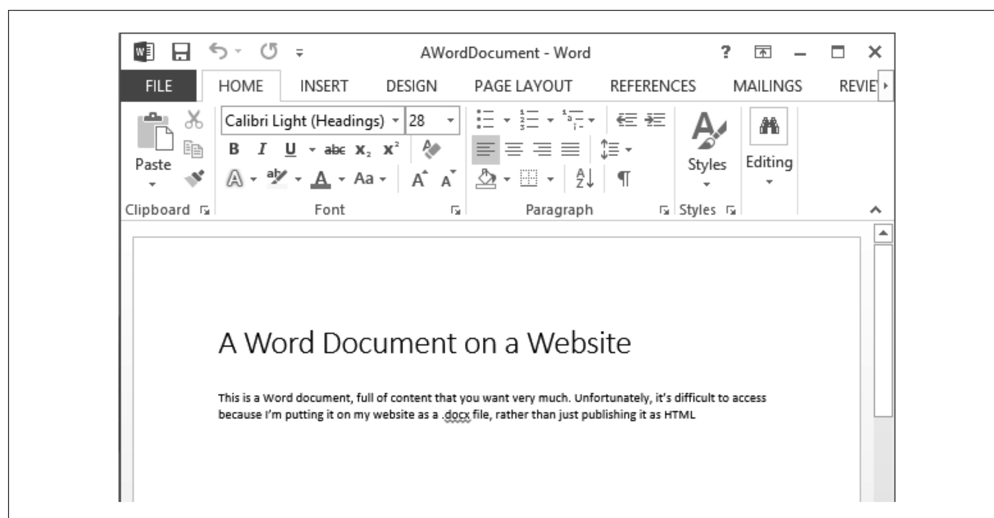


图 7-2：这个 Word 文档的正文内容你可能很想要，但是很难获取，因为我把它放在了网站的 .docx 文件里而不是 HTML 里

上面这个 Python 程序读取这个简单的 Word 文档后，输出的结果如下：

```
<!--?xml version="1.0" encoding="UTF-8" standalone="yes"?-->
<w:document mc:ignorable="w14 w15 wp14" xmlns:m="http://schemas.openx
mlformats.org/officeDocument/2006/math" xmlns:mc="http://schemas.open
xmlformats.org/markup-compatibility/2006" xmlns:o="urn:schemas-micros
oft-com:office:office" xmlns:r="http://schemas.openxmlformats.org/off
iceDocument/2006/relationships" xmlns:v="urn:schemas-microsoft-com:vm
l" xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/m
ain" xmlns:w10="urn:schemas-microsoft-com:office:word" xmlns:w14="htt
p://schemas.microsoft.com/office/word/2010/wordml" xmlns:w15="http://
schemas.microsoft.com/office/word/2012/wordml" xmlns:wne="http://sche
mas.microsoft.com/office/word/2006/wordml" xmlns:wp="http://schemas.o
penxmlformats.org/drawingml/2006/wordprocessingDrawing" xmlns:wp14="h
ttp://schemas.microsoft.com/office/word/2010/wordprocessingDrawing" x
mlns:wpc="http://schemas.microsoft.com/office/word/2010/wordprocessin
gCanvas" xmlns:wpg="http://schemas.microsoft.com/office/word/2010/wor
dprocessingGroup" xmlns:wpi="http://schemas.microsoft.com/office/word
/2010/wordprocessingInk" xmlns:wps="http://schemas.microsoft.com/offi
ce/word/2010/wordprocessingShape"><w:body><w:p w:rsidp="00764658" w:r
sidr="00764658" w:rsidrdefault="00764658"><w:ppr><w:pstyle w:val="Tit
le"></w:pstyle></w:ppr><w:r><w:t>A Word Document on a Website</w:t></
w:r><w:bookmarkstart w:id="0" w:name="_GoBack"></w:bookmarkstart><w:b
ookmarkend w:id="0"></w:bookmarkend></w:p><w:p w:rsidp="00764658" w:r
sidr="00764658" w:rsidrdefault="00764658"></w:p><w:p w:rsidp="0076465
8" w:rsidr="00764658" w:rsidrdefault="00764658" w:rsidrpr="00764658">
<w: r> <w:t>This is a Word document, full of content that you want ve
ry much. Unfortunately, it’s difficult to access because I’m putting
```

```
it on my website as a .</w:t></w:r><w:prooferr w:type="spellStart"></
w:prooferr><w:r><w:t>docx</w:t></w:r><w:prooferr w:type="spellEnd"></
w:prooferr> <w:r> <w:t xml:space="preserve"> file, rather than just p
ublishing it as HTML</w:t> </w:r> </w:p> <w:sectpr w:rsidr="00764658"
 w:rsidrpr="00764658"> <w:pgszw:h="15840" w:w="12240"></w:pgsz><w:pgm
ar w:bottom="1440" w:footer="720" w:gutter="0" w:header="720" w:left=
"1440" w:right="1440" w:top="1440"></w:pgmar> <w:cols w:space="720"><
/w:cols&g; <w:docgrid w:linepitch="360"></w:docgrid> </w:sectpr> </w:
body> </w:document>
```

确实包含了大量的元数据，但是你想要的文本内容被隐藏在 XML 里面。好在文档的所有正文内容都包含在 `w:t` 标签里面，标题内容也是如此，这样就容易处理了。

```
from zipfile import ZipFile
from urllib.request import urlopen
from io import BytesIO
from bs4 import BeautifulSoup

wordFile = urlopen('http://pythonscraping.com/pages/AWordDocument.docx').read()
wordFile = BytesIO(wordFile)
document = ZipFile(wordFile)
xml_content = document.read('word/document.xml')

wordObj = BeautifulSoup(xml_content.decode('utf-8'), 'xml')
textStrings = wordObj.find_all('w:t')

for textElem in textStrings:
    print(textElem.text)
```

注意，这里我们并没有使用此前使用的 BeautifulSoup 的 `html.parser` 解析器，而是使用了 `xml` 解析器。这是因为冒号在 HTML 标签（如 `w:t`）中并不是标准的，而 `html.parser` 不能识别它。

这段代码的结果并不完美，但是已经差不多了。一行打印一个 `w:t` 标签，就可以看到 Word 是如何对文字进行断行处理的：

```
A Word Document on a Website
This is a Word document, full of content that you want very much. Unfortunately,
it’s difficult to access because I’m putting it on my website as a .
docx
 file, rather than just publishing it as HTML
```

你会看到这里“docx”是单独一行，这是因为在原始的 XML 里，它是由 `<w:proofErr w:type="spellStart"/>` 标签包围的。这是 Word 用红色波浪线高亮显示“docx”的方式，提示这个词可能有拼写错误。

文档的标题是由样式定义标签 `<w:pStyle w:val="Title"/>` 处理的。虽然不能非常简单地定位标题（或其他带样式的文本），但是用 BeautifulSoup 的导航功能还是可以帮助我们解决问题的：

```
textStrings = wordObj.find_all('w:t')

for textElem in textStrings:
    style = textElem.parent.parent.find('w:pStyle')
    if style is not None and style['w:val'] == 'Title':
        print('Title is: {}'.format(textElem.text))
    else:
        print(textElem.text)
```

这段代码很容易扩展，以打印不同文本样式的标签，或者把它们标记成其他形式。

第 8 章

数据清洗

到目前为止，我们还没有处理过那些样式不规范的数据，要么是使用样式规范的数据源，要么就是彻底放弃样式不符合预期的数据。但是在网页抓取中，你通常不能对数据源或数据样式太挑剔。

由于存在错误的标点符号、字母大小写不一致、断行和拼写错误等问题，“脏数据”是 Web 上的一个大问题。本章将介绍一些工具和技术，帮助你通过改变代码的编写方式，从源头预防问题，并且对已经进入数据库的数据进行清洗。

8.1　编写代码清洗数据

和写代码处理异常一样，你也应该学习编写预防型代码来处理意外情况。

语言学里有一个模型叫 n-gram，表示文字或语言中 *n* 个连续的单词组成的序列。在进行自然语言分析时，使用 n-gram 或者寻找常用词组，可以很容易地把一句话分解成若干个文字片段。

本节将重点介绍如何获取格式合理的 n-gram，而不用它们做任何分析。在第 9 章，我们再用 2-gram 和 3-gram 来做文本摘要提取和分析。

下面的代码将返回在维基百科词条“Python programming language”中找到的 2-gram 列表：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
```

```
def getNgrams(content, n):
  content = content.split(' ')
  output = []
  for i in range(len(content)-n+1):
    output.append(content[i:i+n])
  return output

html = urlopen('http://en.wikipedia.org/wiki/Python_(programming_language)')
bs = BeautifulSoup(html, 'html.parser')
content = bs.find('div', {'id':'mw-content-text'}).get_text()
ngrams = getNgrams(content, 2)
print(ngrams)
print('2-grams count is: '+str(len(ngrams)))
```

`getNgrams` 函数把一个待处理的字符串分成单词序列（假设所有单词按照空格分开），然后增加到 n-gram 模型（本例中是 2-gram）里形成以每个单词开始的二元数组。

这段程序会从文字中返回一些有意思同时也很有用的 2-gram：

```
['of', 'free'], ['free', 'and'], ['and', 'open-source'], ['open-source', 'softwa
re']
```

不过，同时也会返回一些零乱的数据：

```
['software\nOutline\nSPDX\n\n\n\n\n\n\n\n\nOperating', 'system\nfamilies\n\n\n\n
AROS\nBSD\nDarwin\neCos\nFreeDOS\nGNU\nHaiku\nInferno\nLinux\nMach\nMINIX\nOpenS
olaris\nPlan'], ['system\nfamilies\n\n\n\nAROS\nBSD\nDarwin\neCos\nFreeDOS\nGNU\
nHaiku\nInferno\nLinux\nMach\nMINIX\nOpenSolaris\nPlan', '9\nReactOS\nTUD:OS\n\n
\n\n\n\n\n\nDevelopment\n\n\n\nBasic'], ['9\nReactOS\nTUD:OS\n\n\n\n\n\n\n\n
Development\n\n\n\nBasic', 'For']
```

另外，因为要为每个单词（除了最后一个单词）创建一个 2-gram，所以写作本书时这个词条里共有 7411 个 2-gram。这并不是一个非常便于管理的数据集！

我们首先用一些正则表达式来移除转义字符（如 `\n`），再把 Unicode 字符过滤掉。可以通过下面的函数对之前输出的结果进行清理：

```
import re

def getNgrams(content, n):
    content = re.sub('\n|[[\d+\]]', ' ', content)
    content = bytes(content, 'UTF-8')
    content = content.decode('ascii', 'ignore')
    content = content.split(' ')
    content = [word for word in content if word != '']
    output = []
    for i in range(len(content)-n+1):
        output.append(content[i:i+n])
    return output
```

这将内容中的换行符替换成空格，移除形式如 `[123]` 的引用，过滤掉所有空字符串（由一

行中存在多个空格导致的)。然后，把内容转换成 UTF-8 格式以消除转义字符。

这几步已经可以大大改善输出结果了，但是还有一些问题：

```
['years', 'ago('], ['ago(', '-'], ['-', '-'], ['-', ')'], [')', 'Stable']
```

你可以通过去除每个单词前后的所有标点符号进一步改善结果。这样保留了单词中间的连字符，但是去除了那些在空字符串后面带有一个标点符号的字符串。

当然，标点符号本身是有含义的，简单地将其去除可能会导致丢失一些有价值的信息。例如，一个句点跟着一个空格用来表示一个完整句子的结束。你可能希望 n-gram 中没有跨句子的内容，而是仅包含同一个句子中的内容。

例如，对于下面的文本：

```
Python features a dynamic type system and automatic memory management.
It supports multiple programming paradigms...
```

其中 2-gram `'['memory', 'management']'` 是有效的，而 2-gram `'['management', 'It']'` 则是无效的。

现在“清洗任务”列表变得越来越长，并且你还引入了“句子”的概念，使得你的程序变得更加复杂，因此最好把规则都移出来，创建 4 个不同的函数。

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import string

def cleanSentence(sentence):
    sentence = sentence.split(' ')
    sentence = [word.strip(string.punctuation+string.whitespace)
        for word in sentence]
    sentence = [word for word in sentence if len(word) > 1
        or (word.lower() == 'a' or word.lower() == 'i')]
    return sentence

def cleanInput(content):
    content = re.sub('\n|[[\d+\]]', ' ', content)
    content = bytes(content, "UTF-8")
    content = content.decode("ascii", "ignore")
    sentences = content.split('. ')
    return [cleanSentence(sentence) for sentence in sentences]

def getNgramsFromSentence(content, n):
    output = []
    for i in range(len(content)-n+1):
        output.append(content[i:i+n])
    return output
```

```
def getNgrams(content, n):
    content = cleanInput(content)
    ngrams = []
    for sentence in content:
        ngrams.extend(getNgramsFromSentence(sentence, n))
    return(ngrams)
```

`getNgrams` 仍然是程序的基本切入点。`cleanInput` 像以前一样移除所有的换行符和引用，并且还基于“句点 + 空格”将文本分割成“句子”。程序还调用了 `cleanSentence` 函数，它将句子分割成单词，去除标点符号和空白，还去除除 I 和 a 之外的单字符单词。

创建 n-gram 的关键代码被移动到 `getNgramsFromSentence` 函数中，它在每个句子中通过 `getNgrams` 被调用，这样就保证了 n-gram 不会在句子之间创建。

这里用 `string.punctuation` 和 `string.whitespace` 来获取 Python 所有的标点符号。你可以在 Python 命令行看看 `string.punctuation` 的输出：

```
>>> import string
>>> print(string.punctuation)
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

`print(string.whitespace)` 生成的输出结果不那么有意思（毕竟它只是空白），但是会包括空白字符，如不间断空格、制表符和换行符。

在循环体中用 `item.strip(string.punctuation)` 对内容中的所有单词进行清洗，单词两端的任何标点符号都会被去掉，但带连字符的单词（连字符在单词内部）仍然会保留。

这样输出的 2-gram 结果就更干净了：

```
[['Python', 'Paradigm'], ['Paradigm', 'Object-oriented'], ['Object-oriented',
'imperative'], ['imperative', 'functional'], ['functional', 'procedural'],
['procedural', 'reflective'],...
```

数据标准化

每个人都会遇到一些样式设计不够人性化的网页，比如“请输入你的电话号码。号码格式必须是 xxx-xxx-xxxx”。

作为一名优秀的程序员，你可能会问：“为什么不自动地对输入的信息进行清洗，去掉非数字内容，然后自动给数据加上分隔符呢？”**数据标准化**过程要确保清洗后的数据在语言学或逻辑上是等价的，比如 (555) 123-4567 和 555.123.4567 这两种形式的电话号码实际上是一样的。

还用上一节的 n-gram 示例，让我们在其中增加一些数据标准化特征。

这段代码有一个明显的问题，就是输出结果中包含很多重复的 2-gram 序列。程序把每个

2-gram 序列都加入了列表，没有统计过序列的频率。记录这些 2-gram 序列的频率，而不只是知道某个序列是否存在，这不仅很有意思，而且有助于对比不同的数据清洗和数据标准化算法的效果。如果数据标准化成功了，那么唯一的 n-gram 序列的数量就会减少，而 n-gram 序列的总数（即被认定为 n-gram 的唯一或不唯一的项目的数量）不变。也就是说，对于同样数量的 n-gram 序列，经过去重之后“桶”（bucket）会减少。

你可以修改代码，将 n-gram 结果加入到一个 `Counter` 对象中，而不是列表中：

```
from collections import Counter

def getNgrams(content, n):
    content = cleanInput(content)
    ngrams = Counter()
    for sentence in content:
        newNgrams = [' '.join(ngram) for ngram in
            getNgramsFromSentence(sentence, 2)]
        ngrams.update(newNgrams)
    return(ngrams)
```

当然还有其他实现方式，例如将 n-gram 结果加入到一个字典对象中，其中列表是键，其出现的次数是对应的值。该方法的缺点是它需要更多的管理和排序技巧。但是使用一个 `Counter` 对象也有其缺点：它不能存储列表（因为列表是不可散列的），因此你需要首先在对每个 n-gram 做列表综合时用 `'' '.join(ngram)'` 将列表转换成字符串。

结果如下：

```
Counter({'Python Software': 37, 'Software Foundation': 37, 'of the': 34,
'of Python': 28, 'in Python': 24, 'in the': 23, 'van Rossum': 20, 'to the':
20, 'such as': 19, 'Retrieved February': 19, 'is a': 16, 'from the': 16,
'Python Enhancement': 15,...
```

在写作本书的时候，词条内容一共有 7275 个 2-gram 序列，其中不重复的 2-gram 序列有 5628 个，出现频率最高的 2-gram 序列是“Software Foundation”和“Python Software”。但是，仔细观察结果会发现，“Python Software”还以“Python software”的形式出现两次。同样，“van Rossum”和“Van Rossum”也是作为两个序列统计的。

因此，增加一行代码到 `cleanInput` 函数里：

```
content = content.upper()
```

这样 2-gram 序列的总数还是 7275，而不重复的 2-gram 序列减少到了 5479 个。

除此之外，还需要再考虑一下，自己计划为数据标准化投入多少计算能力。在很多情况下，单词的不同拼写形式其实是等价的，但是为了处理这种等价关系，你需要对每个单词进行检查，以判断它是否和其他单词有等价关系。

比如，“Python 1st”和“Python first”都出现在 2-gram 序列列表里。但是，如果增加一条规则：“让 first、second、third……与 1st、2nd、3rd……等价”，那么每个单词都要额外增加十几次检查。

同理，连字符使用不一致（像“co-ordinated”和“coordinated”）、单词拼写错误以及其他语病（incongruities），都可能对 n-gram 序列的分组结果造成影响，如果语病很严重的话，还可能会彻底打乱输出结果。

对带连字符单词的一种处理方法是，先把连字符去掉，然后把单词当作一个字符串，这只需要一步操作。但是，这样做也会把带连字符的短语（这是很常见的，比如“just-in-time”“object-oriented”等）处理成一个字符串。要是换一种做法，把连字符替换成空格可能更好一点儿。但是你就会见到“co ordinated”和“ordinated attack”之类的 2-gram 序列了！

8.2　数据存储后再清洗

对于编写代码清洗数据，你能做或想做的事情只有这些。除此之外，你可能还需要处理一个由别人创建的数据集，或者一个没见过就不知该如何清洗的数据集。

很多程序员遇到这种情况的自然反应就是“写个脚本”，当然这也是一个很好的解决方法。但是，还有一些第三方工具，像 OpenRefine，不仅可以快速简单地清洗数据，还能让非编程人员轻松地看见和使用你的数据。

OpenRefine

OpenRefine 是 Metaweb 公司在 2009 年启动的一个开源项目。Google 在 2010 年收购了 Metaweb，并把该项目的名称从 Freebase Gridworks 改成了 Google Refine。2012 年，Google 放弃了对 Refine 的支持，让它重新成为开源软件，并将名字改成了 OpenRefine，现在每个人都可以为这个项目做贡献。

1. 安装

OpenRefine 的独特之处在于虽然它的界面运行在浏览器中，但它实际上是一个桌面应用，必须下载并安装。你可以从它的网站下载对应 Linux、Windows 和 macOS 系统的版本。



如果你是 Mac 用户，在打开安装文件的时候遇到了安装权限问题，请到“系统偏好设置→安全性与隐私→通用”，把“允许从以下位置下载应用”的选项设置为“任何来源”。不幸的是，从 Google 项目转变成开源项目之后，OpenRefine 好像在苹果系统中失去了合法性，不再是来源合法的应用程序了。

要想使用 OpenRefine，你需要把数据保存为 CSV 文件（如果你需要了解如何操作，请参考 6.2 节）。另外，如果你的数据已经保存在数据库中，你可以把数据导出为 CSV 文件。

2. 使用OpenRefine

在下面的例子中，我们将使用维基百科的“文本编辑器对比”表格（https://en.wikipedia.org/wiki/Comparison_of_text_editors）里的内容，如图 8-1 所示。虽然这个表格的样式比较规范，但里面包含了多次编辑的痕迹，所以还是有一些样式不一致的地方。另外，因为这个数据是写给人而不是机器看的，所以原来使用的一些样式（比如用“Free”而不是“$0.00”）不太合适作为 OpenRefine 程序的输入数据。

75 rows　Extensions: Freebase

Show as: **rows** records　Show: 5 10 25 **50** rows　« first ‹ previous **1 - 50** next › last »

All	Name	Creator	First public relea	Latest stable ver	Programming language	Cost (US$)	Software license	Open source
1.	Acme	Rob Pike	1993	Plan 9 and Inferno	C	$0	LPL (OSI approved)	Yes
2.	AkelPad	Alexey Kuznetsov, Alexander Shengalts	2003	4.9.0	C	$0	BSD	Yes
3.	Alphatk	Vince Darley	1999	8.3.3		$40	Proprietary, with BSD components	No
4.	Aquamacs	David Reitter	2005	3.0a	C, Emacs Lisp	$0	GPL	Yes
5.	Atom	Github	2014	0.132.0	HTML, CSS, JavaScript, C++	$0	MIT	Yes
6.	BBEdit	Rich Siegel	1992-04	10.5.12	Objective-C, Objective-C++	$49.99	Proprietary	No
7.	Bluefish	Bluefish Development Team	1999	2.2.6	C	$0	GPL	Yes
8.	Coda	Panic	2007	2.0.12	Objective-C	$99	Proprietary	No
9.	ConTEXT	ConTEXT Project Ltd	1999	0.98.6	Object Pascal (Delphi)	$0	BSD	Yes
10.	Crimson Editor	Ingyu Kang, Emerald Editor Team	1999	3.72	C++	$0	GPL	Yes
11.	Diakonos	Pistos	2004	0.9.2	Ruby	$0	MIT	Yes
12.	E Text Editor	Alexander Stigsen	2005	2.0.2		$46.95	Proprietary, with BSD components	No
13.	ed	Ken Thompson	1970	unchanged from original	C	$0	?	Yes
14.	EditPlus	Sangil Kim	1998	3.5	C++	$35	Shareware	No
15.	Editra	Cody Precord	2007	0.6.77	Python	$0	wxWindows license	Yes

图 8-1：显示在 OpenRefine 主屏幕上的维基百科的“文本编辑器对比”表格数据

使用 OpenRefine 时会看到每一列的标签旁边都有一个箭头。这个箭头提供了一个工具菜单，可以对这一列数据执行筛选、排序、变换或删除操作。

筛选。数据筛选可以通过两种方法实现：过滤器（filter）和切片器（facet）。过滤器可以用正则表达式筛选数据，比如“只显示‘Programming language’这一列中包含 3 种或以上用逗号分隔的编程语言的所有行”，结果如图 8-2 所示。

可以通过右边的操作框轻松地组合、编辑和增加过滤器。过滤器还可以和切片器配合使用。

Facet / Filter　Undo / Redo 1

Refresh　Reset All　Remove All

Programming language

.+,.+,.+

case sensitive　regular expression

5 matching rows (75 total)

Show as: **rows** records　Show: 5 10 25 **50** rows

All	Name	Creator	First public re
5.	Atom	Github	201
28.	Komodo Edit	Activestate	open-sourced 2007
29.	Komodo IDE	Activestate	200
59.	Sublime Text	Jon Skinner	200
74.	Zed	Zef Hemel	201

图 8-2：正则表达式“.+,.+,.+”选择至少有 3 种且用逗号分隔的编程语言的所有行

切片器可以很方便地对一列的部分数据进行包含和不包含的筛选（比如，“显示使用 GPL 和 MIT 授权且在 2005 年之后首次发行的所有行”，如图 8-3 所示）。它们都有内置的筛选工具。例如，数值筛选功能会为你提供一个数值滑动条，让你选择需要的数值区间。

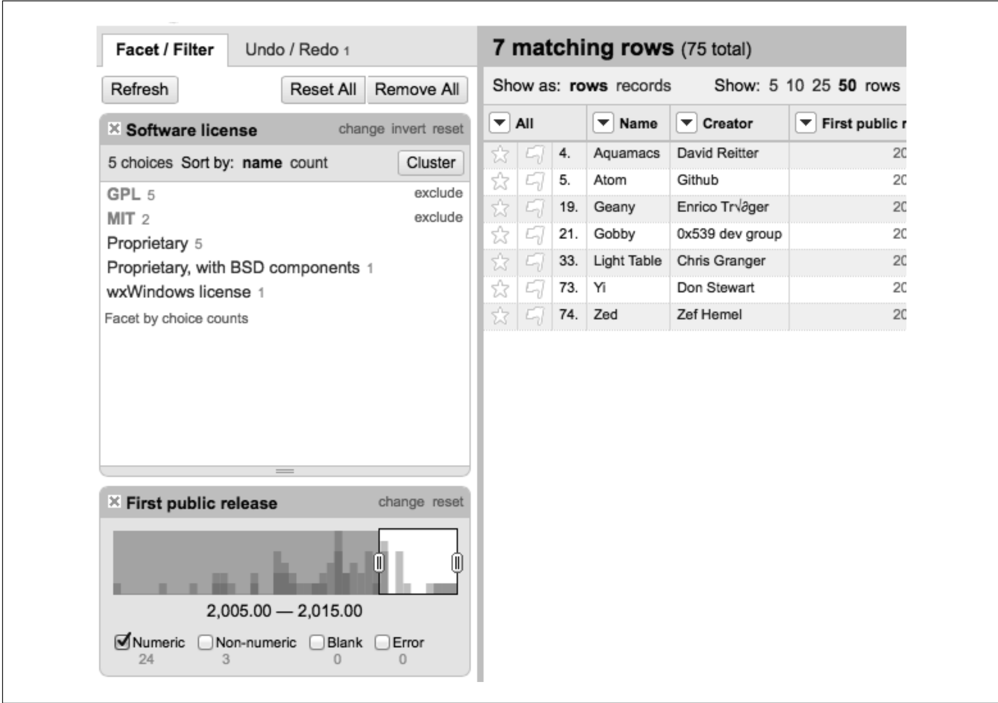


图 8-3：显示使用 GPL 和 MIT 授权且在 2005 年之后首次发行的所有行

筛选后的数据可以导出为 OpenRefine 支持的任意一种数据文件格式，包括 CSV、HTML（HTML 表格）、Excel 以及其他格式。

清洗。只有当数据比较干净时，数据筛选才能成功完成。例如，在前面切片器的例子中，有个文本编辑器的发行日期是“01-01-2006”，而要寻找的数值是“2006”，所以它不能匹配，会被忽略掉，因此在“First public release”切片器中就不会显示了。

OpenRefine 的数据变换功能是通过 OpenRefine 表达式语言（OpenRefine Expression Language，GREL，其中“G”代表 OpenRefine 之前的名字 GoogleRefine）实现的。这种语言通过创建规则简单的 Lambda 函数来实现数据的转换。例如：

```
if(value.length() != 4, "invalid", value)
```

如果把这个函数应用到“First stable release”列，它就只会保留那些“YYYY”形式的数值，把其他数值标记成 `invalid`（无效数据），如图 8-4 所示。

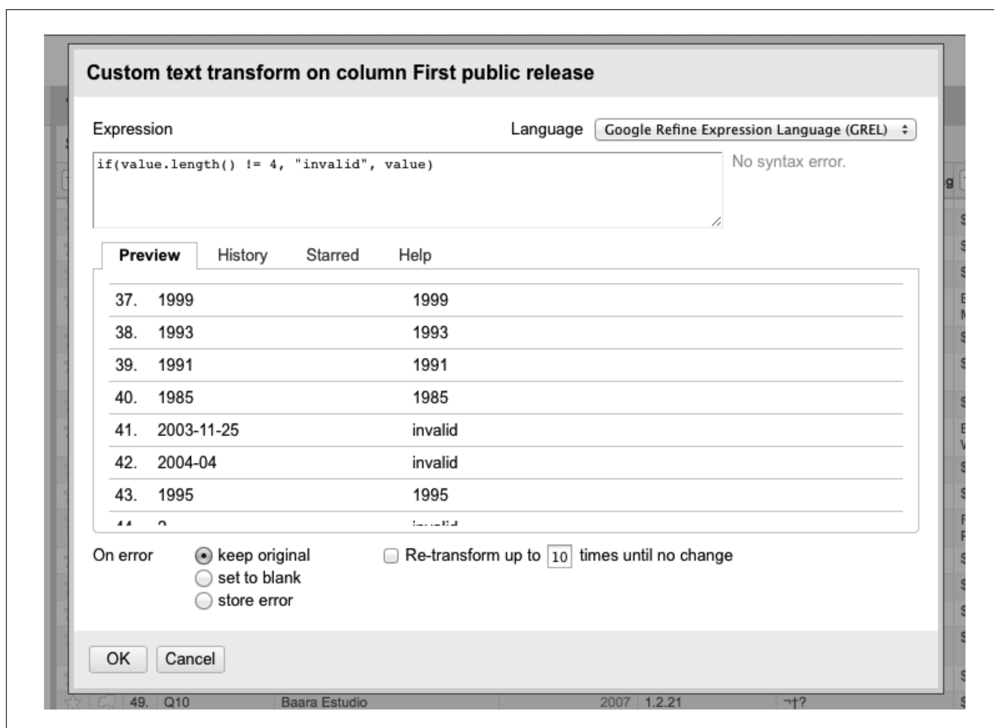


图 8-4：在项目中插入一行 GREL 语句（结果预览显示在语句下面）

点击列标签旁边的向下箭头，再点击“Edit cells”→“Transform”，就可以使用任何 GREL 语句。

但是，把不符合条件的数据标记成无效数据，虽然可以让它们变得容易识别，但是对我们来说用处不大。更好的做法是尽可能地修复那些格式不规范的数据。这可以用 GERL 的 `match` 函数实现：

```
value.match(".*([0-9]{4}).*").get(0)
```

这试图用正则表达式对字符串数据进行匹配。如果正则表达式能够匹配出结果，就会返回一个数组。任何符合正则表达式“捕获组”（capture group）条件的子字符串（指的是括号里的表达式，本例中是 `[0-9]{4}`）都会作为数组数值返回。

其实，这行代码会从一个单元格中找出所有连续的 4 位整数，然后返回第一个匹配结果。一般情况下，这完全可用于从文本或格式不规范的日期数据中提取年份。如果正则表达式没有找到年份，就会返回 `null`。（GREL 在操作 null 变量的时候不会抛出空指针异常。）

通过单元格编辑和 GREL 还可以实现很多其他的数据变换。GREL 的完整指南请参见 OpenRefine 的 GitHub 页面。

第9章

自然语言处理

到目前为止，我们处理的数据大部分都是数字或数值。大多数情况下，我们只是简单地存储数据，没有分析数据。在这一章里，我们将尝试探索英语这个复杂的主题。[1]

当你在 Google 的图片搜索里输入"cute kitten"时，Google 怎么会知道你要搜索什么呢？那是因为可爱小猫咪的图片中常常带有这个词组。当你在 YouTube 搜索框中输入"dead parrot"时，YouTube 怎么会知道要推荐一些 Monty Python 团体的幽默短剧呢？那是因为每个上传的视频里都带有标题和简介文字。

其实，输入"deceased bird monty python"这类短语时，也会立即显示"Dead Parrot"幽默短剧，即使页面本身不包含单词"deceased"或"bird"。Google 知道"hot dog"是一种食物，"boiling puppy"却是另一种完全不同的东西。它究竟是怎么实现的呢？其实这一切都是统计学在起作用！

虽然你可能认为自己的项目和文本分析没有任何关系，但是理解文本分析的原理对各种机器学习场景都是非常有用的，而且还可以提高自己利用概率论和算法知识对现实问题进行建模的能力。

例如，Shazam 音乐雷达是一种可以识别出一段音频中包含哪首歌的服务，即使音频中包

注 1：虽然这一章介绍的很多方法可以用于大多数语种，但是目前只关注英语的自然语言处理是没有问题的。像 Python 的自然语言处理工具包（NLTK）就是面向英语的。互联网上 56% 的内容依然是英文（其次是俄语，只占 6%，http://w3techs.com/technologies/overview/content_language/all）。但是谁知道未来会怎样呢？英语占互联网大头的情况未来几乎肯定会变化，几年后可能就需要更新。

含了环境噪声或失真也没问题。Google 正在实现基本图片本身自动给图片添加说明文字。[2] 比如，通过对比已知的热狗图片和其他热狗图片，搜索引擎就可以不断地学习到热狗的特征，然后对其他图片进行模式识别，从而判断是不是热狗图片。

9.1 概括数据

第 8 章介绍过如何把文本内容分解成 n-gram 模型，或者长度为 *n* 个单词的短语。从基本功能上说，这可以用来确定一段文字中最常用的单词和短语。另外，还可用来从原文中提取包含最常用的短语的句子，从而对原文进行合理的概括。

我们即将用来做数据归纳的文字样本源自美国第九任总统威廉·亨利·哈里森的就职演说。哈里森的总统生涯创下美国总统任职历史的两个记录：一个是最长的就职演说，另一个是最短的任职时间——32 天。

我们将用他的总统就职演说的全文（http://pythonscraping.com/files/inaugurationSpeech.txt）作为这一章许多示例代码的数据源。

简单修改一下我们在第 8 章用过的 n-gram 模型，就可以用来获得 2-gram 序列的频率数据，并返回一个 2-gram 的 `Counter` 对象：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import string
from collections import Counter

def cleanSentence(sentence):
    sentence = sentence.split(' ')
    sentence = [word.strip(string.punctuation+string.whitespace)
        for word in sentence]
    sentence = [word for word in sentence if len(word) > 1
        or (word.lower() == 'a' or word.lower() == 'i')]
    return sentence

def cleanInput(content):
    content = content.upper()
    content = re.sub('\n', ' ', content)
    content = bytes(content, "UTF-8")
    content = content.decode("ascii", "ignore")
    sentences = content.split('. ')
    return [cleanSentence(sentence) for sentence in sentences]

def getNgramsFromSentence(content, n):
    output = []
```

注 2：Oriol Vinyals et al,“A Picture Is Worth a Thousand (Coherent) Words: Building a Natural Description of Images”, Google Research Blog, November 17, 2014.

```
    for i in range(len(content)-n+1):
        output.append(content[i:i+n])
    return output

def getNgrams(content, n):
    content = cleanInput(content)
    ngrams = Counter()
    ngrams_list = []
    for sentence in content:
        newNgrams = [' '.join(ngram) for ngram in
            getNgramsFromSentence(sentence, 2)]
        ngrams_list.extend(newNgrams)
        ngrams.update(newNgrams)
    return(ngrams)


content = str(
      urlopen('http://pythonscraping.com/files/inaugurationSpeech.txt')
        .read(), 'utf-8')
ngrams = getNgrams(content, 2)
print(ngrams)
```

输出结果的一部分是：

```
Counter({'OF THE': 213, 'IN THE': 65, 'TO THE': 61, 'BY THE': 41,
'THE CONSTITUTION': 34, 'OF OUR': 29, 'TO BE': 26, 'THE PEOPLE': 24,
'FROM THE': 24, 'THAT THE': 23,...
```

在这些 2-gram 序列中，“the constitution”像是演说的主旨，“of the”“in the”和“to the”看起来并不重要。怎么才能用准确的方式去掉这些不想要的词呢？

前人已经仔细地研究过这些“有意义的”单词和“没意义的”单词的差异了，他们的工作可以帮助我们完成过滤工作。美国杨百翰大学的语言学教授 Mark Davies 一直在维护当代美式英语语料库（Corpus of Contemporary American English），里面包含了过去 10 多年美国流行出版物中的超过 4.5 亿个单词。

最常用的 5000 个单词列表可以免费获取，作为一个基本的过滤器来过滤最常用的 2-gram 序列绰绰有余。其实只用前 100 个单词就可以大幅改善分析结果，我们增加一个 `isCommon` 函数来实现：

```
def isCommon(ngram):
    commonWords = ['THE', 'BE', 'AND', 'OF', 'A', 'IN', 'TO', 'HAVE', 'IT', 'I',
        'THAT', 'FOR', 'YOU', 'HE', 'WITH', 'ON', 'DO', 'SAY', 'THIS', 'THEY',
        'IS', 'AN', 'AT', 'BUT', 'WE', 'HIS', 'FROM', 'THAT', 'NOT', 'BY',
        'SHE', 'OR', 'AS', 'WHAT', 'GO', 'THEIR', 'CAN', 'WHO', 'GET', 'IF',
        'WOULD', 'HER', 'ALL', 'MY', 'MAKE', 'ABOUT', 'KNOW', 'WILL', 'AS',
        'UP', 'ONE', 'TIME', 'HAS', 'BEEN', 'THERE', 'YEAR', 'SO', 'THINK',
        'WHEN', 'WHICH', 'THEM', 'SOME', 'ME', 'PEOPLE', 'TAKE', 'OUT', 'INTO',
        'JUST', 'SEE', 'HIM', 'YOUR', 'COME', 'COULD', 'NOW', 'THAN', 'LIKE',
        'OTHER', 'HOW', 'THEN', 'ITS', 'OUR', 'TWO', 'MORE', 'THESE', 'WANT',
```

```
        'WAY', 'LOOK', 'FIRST', 'ALSO', 'NEW', 'BECAUSE', 'DAY', 'MORE', 'USE',
        'NO', 'MAN', 'FIND', 'HERE', 'THING', 'GIVE', 'MANY', 'WELL']
    for word in ngram:
        if word in commonWords:
            return True
    return False
```

这样处理之后，就可以得到在样本文字中出现频率不低于 3 次的 2-gram 序列，如下所示：

```
Counter({'UNITED STATES': 10, 'EXECUTIVE DEPARTMENT': 4,
'GENERAL GOVERNMENT': 4, 'CALLED UPON': 3, 'CHIEF MAGISTRATE': 3,
'LEGISLATIVE BODY': 3, 'SAME CAUSES': 3, 'GOVERNMENT SHOULD': 3,
'WHOLE COUNTRY': 3,...
```

效果看着不错，列表中的前两项是"United States"和"executive department"，和我们对总统就职演说的期待是一样的。

这里需要注意的是，我们是用比较新的常用词列表过滤结果的，这对 1841 年写出来的文字来说可能不是非常合适。但是，因为我们只用了列表里的前 100 个单词——我们姑且可以认为，随着年代的变化，这 100 个单词应该比列表最后的 100 个单词更具稳定性——而且也获得了满意的结果，所以好像也不必挖掘或创建一个 1841 年最常用的单词列表（虽然这样的努力可能会很有趣）。

现在一些核心的主题词已经从文本中抽取出来了，它们怎么帮助我们归纳这段文字呢？一种方法是搜索包含每个核心 n-gram 序列的第一句话，这种方法的理论是英语中段落的首句往往是对后面内容的概述。前 5 个 2-gram 序列的搜索结果如下。

- The Constitution of the United States is the instrument containing this grant of power to the several departments composing the government.
- Such a one was afforded by the executive department constituted by the Constitution.
- The general government has seized upon none of the reserved rights of the states.
- Called from a retirement which I had supposed was to continue for the residue of my life to fill the chief executive office of this great and free nation, I appear before you, fellow-citizens, to take the oaths which the constitution prescribes as a necessary qualification for the performance of its duties; and in obedience to a custom coeval with our government and what I believe to be your expectations I proceed to present to you a summary of the principles which will govern me in the discharge of the duties which I shall be called upon to perform.
- The presses in the necessary employment of the government should never be used to clear the guilty or to varnish crime.

当然，这些估计还不能马上发布到 CliffsNotes 上面，但是考虑到全文原来一共有 217 句话，而这里的第四句话（"Called from a retirement..."）已经把主题总结得很好了，作为初稿应该能凑合。

如果是更大段的文本，或者说更复杂的文本，那么当寻找段落中“最重要”的句子时，可能需要看一下 3-gram 甚至是 4-gram 的结果。在本例中只有 3-gram“exclusive metallic currency”被多次使用，而它并不是总统就职演讲中的典型短语。对于更长的段落，使用 3-gram 可能更合适。

另外一种方法是查看包含最常用的 n-gram 的句子。显然，这往往是更长的句子。如果这是个问题的话，你可以寻找常用 n-gram 比例最高的句子，或者自己创建一个评价指标，并综合多种技巧。

9.2 马尔可夫模型

你可能听说过马尔可夫文本生成器。它们因为两种用途而非常受欢迎：娱乐，比如用在 That can be my next tweet! 应用中；用于生成逼真的垃圾邮件来愚弄检测系统。

这些文本生成器都基于马尔可夫模型。马尔可夫模型常用于分析大量的随机事件，其中一个离散事件发生之后，另一个离散事件会以一定的概率发生。

例如，我们可以对一个天气系统建立马尔可夫模型，如图 9-1 所示。

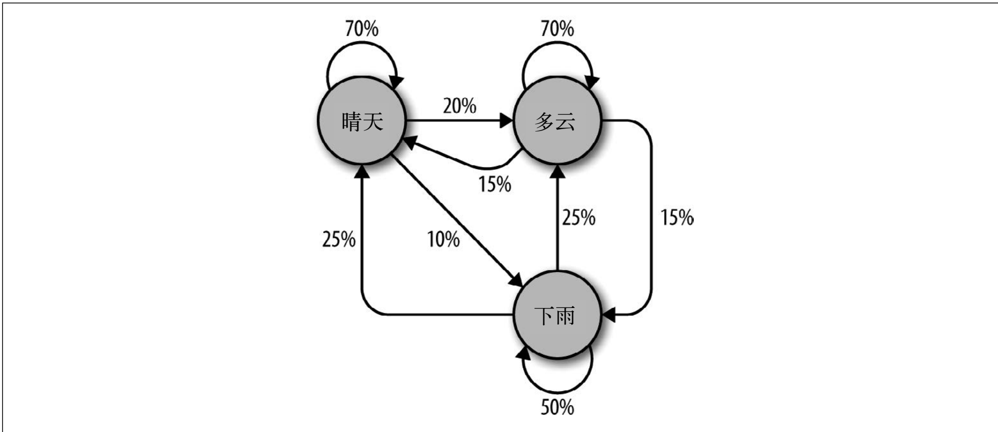


图 9-1：马尔可夫模型描述一个理论上的天气系统

在这个天气系统模型中，如果今天是晴天，那么明天有 70% 的概率是晴天，20% 的概率多云，10% 的概率下雨。如果今天是下雨天，那么明天有 50% 的概率也下雨，25% 的概率是晴天，25% 的概率是多云。

你可能注意到了马尔可夫模型的几个性质。

- 任何一个节点引出的所有概率之和必须等于 100%。无论是多么复杂的系统，必然会在下一步发生若干事件中的一个事件。

- 虽然这个天气系统在任意时刻都只有 3 种可能，但是你可以用这个模型生成一个天气状态的无限列表。
- 只有当前节点的状态会影响后一天的状态。如果你在“晴天”节点上，即使前 100 天都是晴天或雨天也没关系，明天晴天的概率依然是 70%。
- 有些节点可能比其他节点更难到达。这个现象的原因从数学角度来解释非常复杂，但是可以直观地看出，在这个系统中，在任意时间点上，第二天是“雨天”的可能性（指向它的箭头的概率之和小于“100%”）要比“晴天”或“多云”小很多。

很明显，这是一个很简单的系统，而马尔可夫模型可以演化成任意规模的复杂系统。事实上，Google 的 PageRank 算法也是部分基于马尔可夫模型，网站表示为节点，入站 / 出站链接表示为节点之间的连线。连接某一个节点的“可能性”（likelihood）表示一个网站的相对受欢迎程度。也就是说，如果我们的天气系统代表一个微型互联网，那么“雨天”的页面等级（page rank）相对比较低，而“多云”的页面等级相对比较高。

了解了这些概念之后，让我们回到本节的主题，研究一个具体的例子：文本分析与写作。

还用前面例子里分析的威廉 · 亨利 · 哈里森的就职演讲内容，我们可以写出下面的代码，基于文本结构生成任意长度（下面示例中链长为 100）的马尔可夫链。

```
from urllib.request import urlopen
from random import randint

def wordListSum(wordList):
    sum = 0
    for word, value in wordList.items():
        sum += value
    return sum

def retrieveRandomWord(wordList):
    randIndex = randint(1, wordListSum(wordList))
    for word, value in wordList.items():
        randIndex -= value
        if randIndex <= 0:
            return word

def buildWordDict(text):
    # 剔除换行符和引号
    text = text.replace('\n', ' ')
    text = text.replace('"', '')

    # 保证每个标点符号都被当作一个"单词"
    # 这样就不会被剔除，而是会保留在马尔可夫链中
    punctuation = [',','.',';',':']
    for symbol in punctuation:
        text = text.replace(symbol, ' {} '.format(symbol));

    words = text.split(' ')
    # 过滤空单词
```

```
    words = [word for word in words if word != '']

    wordDict = {}
    for i in range(1, len(words)):
        if words[i-1] not in wordDict:
                # 为单词新建一个字典
            wordDict[words[i-1]] = {}
        if words[i] not in wordDict[words[i-1]]:
            wordDict[words[i-1]][words[i]] = 0
        wordDict[words[i-1]][words[i]] += 1
    return wordDict

text = str(urlopen('http://pythonscraping.com/files/inaugurationSpeech.txt')
          .read(), 'utf-8')
wordDict = buildWordDict(text)

# 生成链长为100的马尔可夫链
length = 100
chain = ['I']
for i in range(0, length):
    newWord = retrieveRandomWord(wordDict[chain[-1]])
    chain.append(newWord)

print(' '.join(chain))
```

代码的输出结果每次都会变化，下面是其中一个“胡言乱语”的结果：

```
I sincerely believe in Chief Magistrate to make all necessary sacrifices and
oppression of the remedies which we may have occurred to me in the arrangement
and disbursement of the democratic claims them , consolatory to have been best
political power in fervently commending every other addition of legislation , by
the interests which violate that the Government would compare our aboriginal
neighbors the people to its accomplishment . The latter also susceptible of the
Constitution not much mischief , disputes have left to betray . The maxim which
may sometimes be an impartial and to prevent the adoption or
```

那么代码是怎么实现的呢？

`buildWordDict` 函数把从网上获取的演讲文本的字符串作为参数，然后对字符串进行清理和格式化处理，去掉引号，并在其他标点符号两端加上空格，这样就可以将它们当成一个单独的单词。最后，建立如下所示的一个二维字典——字典里有字典：

```
{word_a : {word_b : 2, word_c : 1, word_d : 1},
word_e : {word_b : 5, word_d : 2},...}
```

在这个字典示例中，“word_a”出现了 4 次，有两次后面跟着“word_b”，一次后面跟着“word_c”，一次后面跟着“word_d”。有 7 个“word_e”后面跟着单词，其中有 5 次后面跟着“word_b”，两次后面跟着“word_d”。

如果我们要画出这个结果的节点模型，那么代表“word_a”的节点将有一个（表示 50% 概

率的）箭头指向“word_b”（在 4 次中，有 2 次是它跟在“word_a”后面），一个（表示 25% 概率的）箭头指向“word_c”，还有一个（表示 25% 概率的）箭头指向“word_d”。

字典创建之后，不管你现在位于文章中的哪个单词之上，都可以将这个字典作为查询表来选择下一个节点。[3] 使用这个二维字典，如果我们现在位于“word_e”节点，那么下一步就要把字典 `{word_b : 5, word_d : 2}` 传给 `retrieveRandomWord` 函数。这个函数会按照字典中单词频次的权重，随机获取一个单词。

通过先确定一个随机的开始词（示例中用的是常见的“I”），我们可以轻易地遍历马尔可夫链，想生成多少单词就生成多少。

当搜集的文本量越大，尤其是来自相似写作风格的数据源时，这些马尔可夫链就越“真实”。尽管这里的例子使用 2-gram 来创建马尔可夫链（即用前一个单词预测下一个单词），但你也可以使用 3-gram 或者更高阶的 n-gram，即用两个或者两个以上的单词预测下一个单词。

在网站抓取中积累的兆字节的文本数据尽管很有意思并且很有用，这样的应用还是很难看出马尔可夫链的实际效果。正如本节前面提到的，马尔可夫链构建的模型是网站如何从一个页面链接到另外一个页面。大量的这些链接可以形成类似网络的图，图的结构非常易于存储、追踪和分析。这样，马尔可夫链就为如何考虑网络抓取以及网络爬虫应该如何思考打下了基础。

维基百科六度分隔：终结篇

在第 3 章，我们创建了一个爬虫来收集从一个维基词条到另一个维基词条的链接（从凯文·贝肯这个词条开始），并在第 6 章将这些链接存储在数据库里。为什么这里又把这个游戏搬出来？因为从一个页面到另一个页面的链接路径选择问题（即找出 https://en.wikipedia.org/wiki/Kevin_Bacon 和 https://en.wikipedia.org/wiki/Eric_Idle 之间的链接路径），与选择一个马尔可夫链（一个单词到另一个单词的路径）是一样的。

这类问题被称为**有向图**（directed graph）问题，其中 A → B 连通，并不意味着 B → A 同样连通。单词“football”后面可能经常跟着单词“player”，但是单词“player”后面却很少跟着单词“football”。虽然凯文·贝肯的维基词条链接到了到他的老家费城（Philadelphia），但是费城的维基百科词条却没有链接回凯文·贝肯。

相反，原来的凯文·贝肯六度分隔游戏是一个**无向图**（undirected graph）问题。例如，凯文·贝肯和朱莉娅·罗伯茨（Julia Roberts）共同出演过电影《别闯阴阳界》（*Flatliners*），

注 3：程序在处理文本中的最后一个单词的下一个节点选择时可能会发生异常，因为这个单词后面没有单词。在我们的例子中，最后一个单词是点号（.），这样会很方便，因为它在文本中一共出现了 215 次，所以选择下一个单词时不会出现问题。但是，在现实工作中，实现一个马尔可夫生成器时，文本的最后一个单词通常是需要慎重考虑的。

因此凯文·贝肯词条会通过《别闯阴阳界》的维基词条链接到朱莉娅·罗伯茨词条，而朱莉娅·罗伯茨词条也会通过《别闯阴阳界》的维基词条链接到凯文·贝肯词条，两者的关系是相互的（就是没有“方向性”）。在计算机科学中，无向图问题没有有向图问题常见，两者都属于计算难题。

虽然解决这两类问题和对应的多个分支问题的方法有很多，但是寻找有向图中最短路径（找出凯文·贝肯的维基百科词条和所有其他词条之间的链接路径）的最佳且最常用的一种方法是**广度优先搜索**（breadth-first search）。

广度优先搜索算法的思路是优先搜寻直接连接到起始页的所有链接（而不是找到一个链接就纵向深入搜索）。如果这些链接不包含目标页面（你想要找的词条），就对第二层链接（通过一个中间页面链接到起始页）进行搜索。这个过程不断重复，直到达到搜索深度限制（本例中使用的层数限制是 6 层）或者找到目标页面为止。

用第 6 章的链接数据表，实现一个完整的广度优先搜索算法，代码如下所示。

```
import pymysql

conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
    user='', passwd='', db='mysql', charset='utf8')
cur = conn.cursor()
cur.execute('USE wikipedia')

def getUrl(pageId):
    cur.execute('SELECT url FROM pages WHERE id = %s', (int(pageId)))
    return cur.fetchone()[0]

def getLinks(fromPageId):
    cur.execute('SELECT toPageId FROM links WHERE fromPageId = %s',
        (int(fromPageId)))
    if cur.rowcount == 0:
        return []
    return [x[0] for x in cur.fetchall()]

def searchBreadth(targetPageId, paths=[[1]]):
    newPaths = []
    for path in paths:
        links = getLinks(path[-1])
        for link in links:
            if link == targetPageId:
                return path + [link]
            else:
                newPaths.append(path+[link])
    return searchBreadth(targetPageId, newPaths)

nodes = getLinks(1)
targetPageId = 28624
pageIds = searchBreadth(targetPageId)
for pageId in pageIds:
    print(getUrl(pageId))
```

这里函数 getUrl 是辅助函数，用来通过给定的页面 ID 从数据库获取 URL 链接。类似地，getLinks 以 fromPageId（表示当前页面的整数 ID）为输入参数，获取该页面链接到的所有页面的整数 ID 列表。

主函数 searchBreadth 会递归地从搜索页面开始构建所有可能的路径列表，并在找到一个已到达目标页面的路径时停止。

- 它从单个路径 [1] 开始。用户停留在 ID 为 1 的目标页面（Kevin Bacon），并且没有进一步的链接了。
- 对于路径列表中的每一条路径（对于第一次循环，只有一条路径，因此这一步很简短），它会获取所有从该页面（表示为路径中的最后一个页面）链出的链接。
- 对于每个链出的链接，它都会检查其是否与 targetPageId 匹配。如果匹配上了，则返回该路径。
- 如果没有匹配，那么会将一条新的路径添加进新的路径列表（现在变长了），该新的路径列表由旧路径和新的链出路径组成。
- 如果在当前层级没有找到 targetPageId，那么程序就会用 targetPageId 和新的更长的路径列表，递归调用 searchBreadth。

找到页面 ID 列表（包含两个页面之间的路径）后，每个 ID 会对应到其实际的 URL 链接并打印出来。

下面是凯文·贝肯词条（在数据库中页面 ID 为 1）和埃里克·艾德尔词条（在数据库中页面 ID 为 28624）的链接路径：

```
/wiki/Kevin_Bacon
/wiki/Primetime_Emmy_Award_for_Outstanding_Lead_Actor_in_a_
Miniseries_or_a_Movie
/wiki/Gary_Gilmore
/wiki/Eric_Idle
```

链接之间的关系是：Kevin Bacon → Primetime Emmy Award → Gary Gilmore → Eric Idle。

除了解决“六度分隔”问题以及对句子中一个单词后面跟着哪个单词进行建模，有向图和无向图还可用于对网页抓取中的许多场景进行建模。例如，哪个网站链接到了哪个网站？哪篇学术论文引用了其他的学术论文？零售网站上哪些产品往往一并展示？这个链接的强度是什么？这个链接是双向链接吗？

了解这些基本的关系类型对建模、可视化以及基于抓取数据进行预测都非常有用。

9.3　自然语言工具包

到目前为止，本章主要讨论了对文本中单词的统计分析。哪些单词使用得最频繁？哪些单词用得少？一个单词后面可能跟着哪几个单词？这些单词是如何组合在一起的？我们还没

有理解每个单词的具体含义。

自然语言工具包（Natural Language Toolkit，NLTK）是一个 Python 库，用于识别和标记英语文本中单词的词性。这个项目于 2000 年创建，在过去的十多年里，由来自世界各地的几十个开发者共同努力维护。虽然它的功能非常丰富（有几本书专门介绍了 NLTK），但本节只介绍它的几种用法。

9.3.1 安装与设置

`nltk` 模块的安装方法和其他 Python 模块一样，要么从 NLTK 网站直接下载安装包进行安装，要么用第三方安装程序通过关键词“nltk”搜索安装。详细的安装教程，请参考 NLTK 网站。

模块安装之后，可以下载 NLTK 自带的文本库，这样你就可以非常轻松地试用 NLTK 的功能。在 Python 命令行中输入下面的命令即可：

```
>>> import nltk
>>> nltk.download()
```

这两行命令会打开 NLTK 的下载器（见图 9-2）。

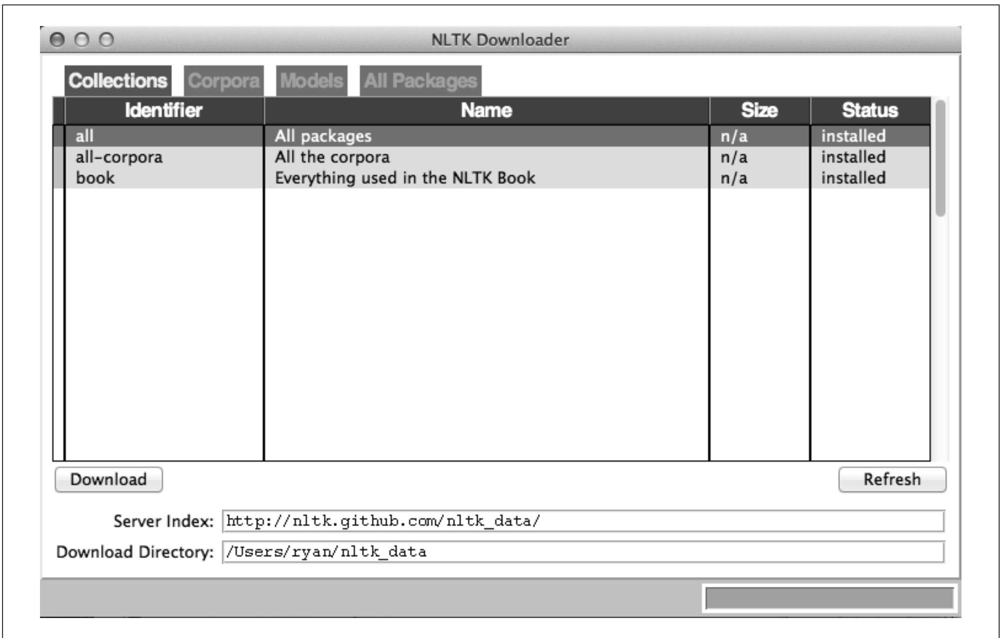


图 9-2：NLTK 下载器可以让你浏览和下载 `nltk` 模块的包和文本库

建议你在刚开始试用 NLTK 语料库时安装所有可用的包。你随时可以轻松卸载这些包。

9.3.2 用NLTK做统计分析

NLTK 很擅长生成统计信息，包括对一段文字的单词数量、单词频率和单词词性进行统计。如果你只需要做一些非常简单的计算（比如计算一段文字中不重复的单词的数量），导入 nltk 模块就太大材小用了——它是一个非常大的模块。但是，如果你需要对文本做更复杂的分析，那么里面有许多函数可以帮你实现任何统计指标。

用 NLTK 做统计分析一般是从 Text 对象开始的。Text 对象可以通过下面的方法用简单的 Python 字符串来创建：

```
from nltk import word_tokenize
from nltk import Text

tokens = word_tokenize('Here is some not very interesting text')
text = Text(tokens)
```

word_tokenize 函数的参数可以是任何 Python 文本字符串。如果你手边没有任何长字符串，但是还想尝试一些功能，NLTK 库里内置了几本书，可以用 import 函数导入：

```
from nltk.book import *
```

这样会加载 9 本书：

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

在后面的例子，我们都用 text6，“Monty Python and the Holy Grail”（一部 1975 年的电影的剧本）。

文本对象可以像普通的 Python 数组那样操作，就好像它们是一个包含文本里所有单词的数组。利用这个属性，你可以统计文本中不重复的单词，然后与总单词数进行比较（还记得 Python 的 set 只保留唯一的值吧）：

```
>>> len(text6)/len(set(text6))
7.833333333333333
```

前面的数据表明剧本中每个单词平均被使用了 8 次。你还可以将文本对象放到一个频率分

布对象 FreqDist 中，查看哪些单词是最常用的，以及各个单词的频率是多少。

```
>>> from nltk import FreqDist
>>> fdist = FreqDist(text6)
>>> fdist.most_common(10)
[(':', 1197), ('.', 816), ('!', 801), (',', 731), ("'", 421), ('[', 3
19), (']', 312), ('the', 299), ('I', 255), ('ARTHUR', 225)]
>>> fdist["Grail"]
34
```

因为这是一个剧本，所以剧本中创作的一些角色会显示出来。例如，全部大写的“ARTHUR”频繁地出现，因为它会出现在亚瑟王（King Arthur）每一句台词的前面。另外，分号（:）也出现在每一行的开头，作为分隔符把人物的姓名和人物的台词分开。根据这个特征，我们可以看到这个电影剧本一共有 1197 句台词！

前几章中用过的 2-gram 模型，在 NLTK 中称作 bigrams（你可能会听到有人把 3-gram 模型叫作“trigrams”，我个人更喜欢用 2-gram 和 3-gram 而不是 bigrams 或 trigrams）。你可以用 NLTK 非常轻松地创建、搜索和列出 2-gram：

```
>>> from nltk import bigrams
>>> bigrams = bigrams(text6)
>>> bigramsDist = FreqDist(bigrams)
>>> bigramsDist[('Sir', 'Robin')]
18
```

为了搜索 2-gram 序列“Sir Robin”，我们需要把它分解成一个元组 ("Sir", "Robin")，用来匹配这个 2-gram 序列在频率分布中的表现方式。还有一个 trigrams 模块，它的工作方式完全相同。对于一般的情形，你还可以导入 ngrams 模块：

```
>>> from nltk import ngrams
>>> fourgrams = ngrams(text6, 4)
>>> fourgramsDist = FreqDist(fourgrams)
>>> fourgramsDist[('father', 'smelt', 'of', 'elderberries')]
1
```

ngrams 函数被用来把文本对象分解成任意规模的 n-gram 序列，第二个参数决定了规模的大小。在这个例子中，我把文本分解成了 4-gram。然后，我就可以查出“father smelt of elderberries”这个短语在剧本中只出现了一次。

频率分布、文本对象和 n-gram 还可以整合到一个循环中进行迭代。例如，下面的程序会打印出文本中所有以“coconut”开头的 4-gram 序列：

```
from nltk.book import *
from nltk import ngrams
fourgrams = ngrams(text6, 4)
for fourgram in fourgrams:
    if fourgram[0] == 'coconut':
        print(fourgram)
```

NLTK 库中有许多不同的工具和对象，用于对大段文字进行组织、统计、排序和度量。尽管我们只是了解了 NLTK 函数用法的皮毛，但是大多数工具都设计得非常好，熟悉 Python 的人很容易操作它们。

9.3.3　用NLTK做词性分析

到现在为止，我们只是基于拼写方式对比和分类遇到的所有单词，并没有区分同形同音异义词或语境。

虽然有人可能会认为同形同音异义词不处理也基本没什么问题，但是如果你看到了它们的使用频率，可能会吓一跳。大多数以英语为母语的人往往不会注意到一个单词是同形同音异义词，也不认为同一个词在不同的语境中会导致意思混乱。

“He was objective in achieving his objective of writing an objective philosophy, primarily using verbs in the objective case”（在实现写作一本客观哲学书的目标时，他是客观的，因为他在描述客观情况时主要使用动词）这句话很容易被人类理解，但是网络爬虫可能会认为同一个单词（objective）被用了 4 次，进而简单地忽略这 4 个单词各自的含义。

除了理清词性，区分出一个单词的用法也有帮助。例如，你可能需要查找一些由普通英文单词组成的公司名称，或者分析某个人对一个公司的评价。“ACME Products is good”和“ACME Products is not bad”意思是一样的，即使一句话里用的是“good”，而另一句话用的是“bad”。

Penn Treebank 语义标记

NLTK 默认使用的是由美国宾夕法尼亚大学大学 Penn Treebank 项目开发的一个流行的词性标注系统。虽然有些标记意思明确（比如，CC 就是并列连接词，coordinating conjunction），有些标记却比较模糊（比如，RP 是小品词，particle）。下面是语义标记的对照表。

缩写	全称
CC	并列连接词（coordinating conjunction）
CD	基数（cardinal number）
DT	限定词（determiner）
EX	表示存在性的“there”（existential “there”）
FW	外来语（foreign word）
IN	介词，从属连词（preposition, subordinating conjunction）
JJ	形容词（adjective）
JJR	形容词，比较级（adjective, comparative）
JJS	形容词，最高级（adjective, superlative）
LS	列表项标记符（list item marker）

MD	情态动词（modal）
NN	名词，单数或不可数（noun, singular or mass）
NNS	名词，复数（noun, plural）
NNP	专有名词，单数（proper noun, singular）
NNPS	专有名词，复数（proper noun, plural）
PDT	前置限定词（predeterminer）
POS	名词所有格 s 结尾（possessive ending）
PRP	人称代词（personal pronoun）
PRP$	物主代词（possessive pronoun）
RB	副词（adverb）
RBR	副词，比较级（adverb, comparative）
RBS	副词，最高级（adverb, superlative）
RP	小品词（particle）
SYM	符号（symbol）
TO	介词"to"（"to"）
UH	感叹词（interjection）
VB	动词，一般形式（verb, base form）
VBD	动词，过去时（verb, past tense）
VBG	动词，动名词或现在分词（verb, gerund or present participle）
VBN	动词，过去分词（verb, past participle）
VBP	动词，非第三人称单数（verb, non-third person singular present）
VBZ	动词，第三人称单数（verb, third person singular present）
WDT	Wh- 限定词（wh-determiner）
WP	Wh- 代词（wh-pronoun）
WP$	Wh- 物主代词（possessive wh-pronoun）
WRB	Wh- 副词（wh-adverb）

除了度量语言，NLTK 还可以基于语境和它的超级大字典分析文本内容，帮助人们寻找单词的含义。NLTK 的一个基本功能是识别句子中各个单词的词性：

```
>>> from nltk.book import *
>>> from nltk import word_tokenize
>>> text = word_tokenize('Strange women lying in ponds distributing swords'\
'is no basis for a system of government.')
>>> from nltk import pos_tag
>>> pos_tag(text)
[('Strange', 'NNP'), ('women', 'NNS'), ('lying', 'VBG'), ('in', 'IN')
, ('ponds', 'NNS'), ('distributing', 'VBG'), ('swords', 'NNS'), ('is'
, 'VBZ'), ('no', 'DT'), ('basis', 'NN'), ('for', 'IN'), ('a', 'DT'),
('system', 'NN'), ('of', 'IN'), ('government', 'NN'), ('.', '.')]
```

每个单词被放在一个**元组**中，一边是单词，一边是 NLTK 的词性标记（每个词性标记的具体含义，请参考前面的 Penn Treebank 标记表）。虽然这看起来是非常简单的查询，但是要

正确地完成这项任务其实很复杂，下面的例子明显体现出了这一点。

```
>>> text = word_tokenize('The dust was thick so he had to dust')
>>> pos_tag(text)
[('The', 'DT'), ('dust', 'NN'), ('was', 'VBD'), ('thick', 'JJ'), ('so
', 'RB'), ('he', 'PRP'), ('had', 'VBD'), ('to', 'TO'), ('dust', 'VB')]
```

需要注意的是，“dust”在这句话里出现了两次：一次是名词，而另一次是动词。NLTK 可以基于句子的内容正确地识别出相应的用法。NLTK 用英语的**上下文无关文法**（context-free grammar）识别词性。上下文无关文法基本上可以看成一个规则集合，用一个有序的列表确定一个词后面可以跟哪些词。NLTK 的上下文无关文法定义的是一个词性后面可以跟哪些词性。无论什么时候，只要遇到像“dust”这样一个含义不明确的单词，NLTK 都会用上下文无关文法的规则来判断，然后确定一个合适的词性。

机器学习和机器训练

你也可以对 NLTK 进行训练，使它针对一门外语创建出一个全新的上下文无关文法规则。如果你用 Penn Treebank 词性标记，手工对该语言的若干大段文本做了语义标记，那么你就可以把结果传给 NLTK，然后训练它对其他文本进行语义标记。在任何一个机器学习场景中，机器训练都是不可或缺的一部分，我们在第 13 章训练爬虫识别验证码（CAPTCHA）时会再做介绍。

那么，知道某段文字中一个词是动词还是名词有什么用呢？这对于在计算机科学研究室里做研究可能有用，但是它对网页抓取有什么用呢？

在网页抓取中经常需要处理搜索的问题。你在抓取了一个网站的文字之后，可能想从文字里面搜索“google”这个词，但你要的是作为动词的 google，而不要作为专有名词的 Google。或者你就想查找 Google 公司的名称 Google，但是不想通过首字母大写来找出答案（人们可能忘记将首字母大写，直接写成了 google）。这时函数 `pos_tag` 就很管用了：

```
from nltk import word_tokenize, sent_tokenize, pos_tag
sentences = sent_tokenize('Google is one of the best companies in the world.'\
' I constantly google myself to see what I\'m up to.')
nouns = ['NN', 'NNS', 'NNP', 'NNPS']

for sentence in sentences:
    if 'google' in sentence.lower():
        taggedWords = pos_tag(word_tokenize(sentence))
            for word in taggedWords:
                if word[0].lower() == 'google' and word[1] in nouns:
                    print(sentence)
```

这段代码只会打印包含名词（而非动词）“google”或“Google”的句子。当然，你也可以明确地要求只打印标记为“NNP”（专有名词）的“Google”，但是 NLTK 有时也会判断错误，所以最好还是根据情况给自己留一些回旋的余地。

自然语言中的许多歧义问题都可以用 NLTK 的 `pos_tag` 函数解决。不只是搜索目标单词或短语，而是搜索带标记的目标单词或短语，这样可以大大提高爬虫搜索的准确率和有效性。

9.4 其他资源

利用机器处理、分析和理解自然语言是计算机科学中最难的任务之一，关于这个主题已有数不清的专著和学术论文。希望本章内容可以让你将思路扩展至传统的网页抓取之外，至少在从事需要进行自然语言分析的项目时，清楚应该从哪儿下手。

关于自然语言处理和 Python 的 NLTK 有许多非常优秀的学习资源。尤其是 Steven Bird、Ewan Klein 和 Edward Loper 合著的 *Natural Language Processing with Python* 对这个主题进行了全面的基础性介绍。

另外，James Pustejovsky 和 Amber Stubbs 合著的 *Natural Language Annotation for Machine Learning* 为自然语言处理提供了更高级的理论指导。学习该书需要有 Python 基础，书中介绍的主题都可以用 Python 的 NLTK 完美地实现。

第10章

穿越网页表单与登录窗口进行抓取

掌握了网页抓取的基础知识之后，你首先遇到的一个问题是："我怎么获取登录窗口背后的信息呢？"如今，Web 正在朝着页面交互、社交媒体、用户生成内容的趋势不断地演进。表单和登录窗口是许多网站中不可或缺的组成部分。不过，它们还是比较容易处理的。

到目前为止，本书示例中的网络爬虫在和 Web 服务器进行数据交互时，基本都是用 HTTP 协议的 `GET` 方法去请求信息。这一章，我们将重点介绍 `POST` 方法，即把信息推送到 Web 服务器进行存储和分析。

表单基本上可以看成一种用户提交 `POST` 请求的方式，且这种请求方式是 Web 服务器能够理解和使用的。就像网站的链接标记可以帮助用户发出 `GET` 请求一样，HTML 表单可以帮助用户发出 `POST` 请求。当然，我们也可以写一点儿代码来自己创建这些请求，然后通过网络爬虫把它们提交给服务器。

10.1　Python Requests库

虽然用 Python 的标准库就可以应对网页表单，但是有时用一点儿语法糖可以让生活更甜蜜。当你想做的不只是用 `urllib` 库实现基本的 `GET` 请求时，可以看看 Python 标准库之外的第三方库。

Requests 库就是一个擅长处理复杂的 HTTP 请求、cookie、header（响应头和请求头）等内容的 Python 第三方库。下面是 Requests 的创建者 Kenneth Reitz 对 Python 标准库工具的评价：

> Python 的标准库 urllib2 为你提供了大多数 HTTP 功能，但是它的 API 非常差劲。它是为当时的 Web 创建的。即便是为了完成最简单的任务，它也需要大量的工作（甚至要重写整个方法）。
>
> 事情不应该这样复杂，在 Python 里更不应该如此。

和任何 Python 第三方库一样，Requests 库也可以用其他第三方 Python 库管理器（比如 pip）安装，或者直接下载源代码安装。

10.2 提交一个基本表单

大多数网页表单都是由一些 HTML 字段、一个提交按钮和一个进行表单处理的操作页面构成的。虽然这些 HTML 字段通常由文字内容构成，但是也可以实现文件上传或包含其他非文字内容。

因为大多数主流网站都会在它们的 robots.txt 文件里注明禁止爬虫接入登录表单（第 18 章介绍了抓取这类表单的相关法律责任），所以为了安全起见，我在 pythonscraping.com 网站里构建了一组不同类型的表单和登录窗口，以便你用网络爬虫抓取。最简单的表单位于 http://pythonscraping.com/pages/files/form.html。

这个表单的源代码是：

```
<form method="post" action="processing.php">
First name: <input type="text" name="firstname"><br>
Last name: <input type="text" name="lastname"><br>
<input type="submit" value="Submit">
</form>
```

这里有几点需要注意一下。首先，两个输入字段的名称是 `firstname` 和 `lastname`，这一点非常重要。这两个字段的名称决定了表单提交后要被 `POST` 到服务器上的可变参数的名称。如果你想模拟表单提交数据的行为，就要保证你的变量名称与字段名称是一一对应的。

其次，表单的操作发生在 processing.php（绝对路径是 http://pythonscraping.com/files/processing.php）。对表单的任何 `POST` 请求其实都发生在这个页面上，而非表单本身所在的页面。切记：HTML 表单的目的，只是帮助网站的访问者将格式正确的请求发送到进行实际操作的页面。除非你要对请求的格式进行研究，否则不需要花太多时间在表单所在的页面上。

用 Requests 库提交表单只需 4 行代码，包括导入库文件的语句和打印内容的指令（是的，就是这么简单）：

```
import requests

params = {'firstname': 'Ryan', 'lastname': 'Mitchell'}
r = requests.post("http://pythonscraping.com/pages/processing.php", data=params)
print(r.text)
```

表单被提交之后，程序应该会返回页面的内容：

```
Hello there, Ryan Mitchell!
```

这个程序还可以用于许多网站的简单表单。比如 O'Reilly Media 新闻订阅页面的表单源代码如下所示：

```
<form action="http://post.oreilly.com/client/o/oreilly/forms/
              quicksignup.cgi" id="example_form2" method="POST">
    <input name="client_token" type="hidden" value="oreilly" />
    <input name="subscribe" type="hidden" value="optin" />
    <input name="success_url" type="hidden" value="http://oreilly.com/store/
                newsletter-thankyou.html" />
    <input name="error_url" type="hidden" value="http://oreilly.com/store/
                newsletter-signup-error.html" />
    <input name="topic_or_dod" type="hidden" value="1" />
    <input name="source" type="hidden" value="orm-home-t1-dotd" />
    <fieldset>
        <input class="email_address long" maxlength="200" name=
                    "email_addr" size="25" type="text" value=
                    "Enter your email here" />
        <button alt="Join" class="skinny" name="submit" onclick=
                      "return addClickTracking('orm','ebook','rightrail','dod'
                                                         );" value="submit">Join</button>
    </fieldset>
</form>
```

虽然乍看会觉得恐怖，但是大多数情况下（后面会介绍异常）你只需要关注两件事：

- 你想提交数据的字段的名称（在这个例子中是 `email_addr`）
- 表单的 `action` 属性，也就是表单提交后网站会显示的页面（在这个例子中是 http://post.oreilly.com/client/o/oreilly/forms/quicksignup.cgi）

添加所需信息，然后运行代码即可：

```
import requests
params = {'email_addr': 'ryan.e.mitchell@gmail.com'}
r = requests.post("http://post.oreilly.com/client/o/oreilly/forms/quicksignup.cgi",
                   data=params)
print(r.text)
```

在这个示例中，你真正加入 O'Reilly 的邮件列表之前，还要填写另一个表单，同样的概念也适用于该表单。不过，如果你自己在家做，希望你慎用这些知识，不要给 O'Reilly 出版社提交很多无效的注册。

10.3　单选按钮、复选框和其他输入

显然，并非所有的网页表单都只是一堆文本字段和一个提交按钮。HTML 标准里提供了大量可用的表单输入字段：单选按钮、复选框和下拉选框等。HTML5 还增加了其他的控件，比如滚动条（范围输入字段）、邮箱、日期等。自定义的 JavaScript 字段可谓无所不能，可

以实现取色器（colorpicker）、日历以及开发者能想到的任何功能。

无论表单的字段看起来多么复杂，仍然只有两件事是需要关注的：字段名称和字段值。字段名称可以通过查看源代码并寻找 `name` 属性轻易获得。而字段的值有时会比较复杂，因为它有可能是在表单提交之前通过 JavaScript 生成的。取色器就是一个比较奇怪的表单字段，它可能会用类似 `#F03030` 这样的值。

如果你不确定输入字段值的数据格式，有一些工具可以跟踪浏览器和网站之间来回发送的 `GET` 和 `POST` 请求。前面提过，跟踪 `GET` 请求最显而易见的方式就是看网站的 URL 链接。如果 URL 链接像这样：

```
http://domainname.com?thing1=foo&thing2=bar
```

那么你就知道这个请求对应下面这种表单：

```
<form method="GET" action="someProcessor.php">
<input type="someCrazyInputType" name="thing1" value="foo" />
<input type="anotherCrazyInputType" name="thing2" value="bar" />
<input type="submit" value="Submit" />
</form>
```

对应的 Python 参数是：

```
{'thing1':'foo', 'thing2':'bar'}
```

如果你遇到了一个看起来很复杂的 `POST` 表单，并且想查看浏览器向服务器传递了哪些参数，最简单的方法就是用浏览器的检查器（inspector）或开发者工具查看，如图 10-1 所示。

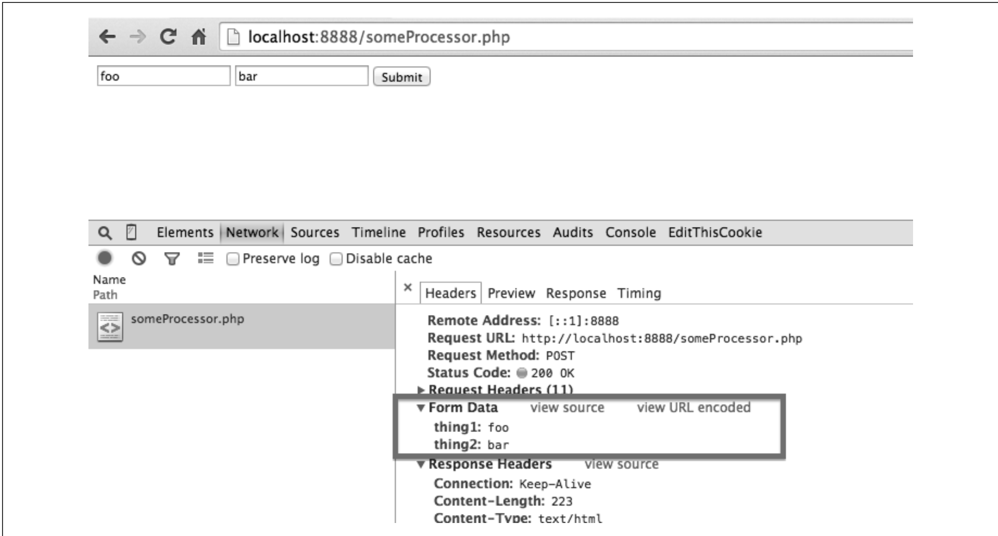


图 10-1：方框高亮的 Form Data 部分显示了 `POST` 请求参数“thing1”和“thing2”，以及对应的值“foo”和“bar”

Chrome 浏览器的开发者工具可以在菜单中通过“更多工具”→“开发者工具”打开（快捷键 F12）。它提供了浏览器与网站交互时产生的所有请求，是一种详细查看请求参数的好方法。

10.4　提交文件和图像

虽然文件上传在网络上很普遍，但是在网页抓取中却不太常用。但是，如果你想为自己网站的文件上传功能写一个测试实例，也是可以实现的。不管怎么说，掌握工作原理总是有用的。

http://pythonscraping.com/files/form2.html 上有一个文件上传表单。页面上表单的源代码如下所示：

```
<form action="processing2.php" method="post" enctype="multipart/form-data">
  Submit a jpg, png, or gif: <input type="file" name="uploadFile"><br>
  <input type="submit" value="Upload File">
</form>
```

除了 `<input>` 标签里有一个 `type` 属性是 `file`，文件上传表单看起来和前面例子中的基于文本的表单没什么两样。其实，Python Requests 库对这种表单的处理方式也和之前的非常相似：

```
import requests

files = {'uploadFile': open('files/python.png', 'rb')}
r = requests.post('http://pythonscraping.com/pages/processing2.php',
                  files=files)
print(r.text)
```

需要注意，这里提交给表单字段 `uploadFile` 的值不是一个简单的字符串了，而是一个由 `open` 函数返回的 Python 文件对象。在这个例子中，我提交了一个保存在我电脑上的图像文件，文件路径是相对这个 Python 程序所在位置的 ../files/Python-logo.png。

没错，就是这么简单！

10.5　处理登录和cookie

到此为止，我们主要讨论的是允许你向网站提交信息的表单，或者在提交后能立即在页面上看到所需信息的表单。那么，这些表单和登录表单（当你浏览网站时让你保持“已登录”状态）有什么不同？

大多数现代网站都用 cookie 跟踪用户是否已登录的状态信息。一旦网站验证了你的登录凭据，就会在你的浏览器上将其保存为一个 cookie，里面通常包含一个由服务器生成的令牌、登录有效时限和状态跟踪信息。网站会把这个 cookie 当作一种验证证据，在你浏览网

站的每个页面时都出示给服务器。在 20 世纪 90 年代中期广泛使用 cookie 之前，保证用户安全验证并跟踪用户对网站来说是一个大问题。

虽然 cookie 为 Web 开发者解决了大问题，却会给网络爬虫带来问题。你可以一整天只提交一次登录表单，但是如果你没有跟踪表单后来回传给你的那个 cookie，那么一段时间以后你访问新页面时，你的登录状态就会丢失，你需要重新登录。

我在 http://pythonscraping.com/pages/cookies/login.html 创建了一个简单的登录表单（用户名可以是任意值，但密码必须是"password"）。这个表单在欢迎页面（http://pythonscraping.com/pages/cookies/welcome.php）处理，里面包含一个到主面的链接：http://pythonscraping.com/pages/cookies/profile.php。

如果在登录网站之前你试图访问欢迎页面或者简介页面，会看到一个错误信息和请先登录的指令。在简介页面中，网站会检测浏览器的 cookie，看它有没有页面已登录的设置信息。

用 Requests 库跟踪 cookie 同样很简单：

```
import requests

params = {'username': 'Ryan', 'password': 'password'}
r = requests.post('http://pythonscraping.com/pages/cookies/welcome.php', params)
print('Cookie is set to:')
print(r.cookies.get_dict())
print('Going to profile page...')
r = requests.get('http://pythonscraping.com/pages/cookies/profile.php',
                 cookies=r.cookies)
print(r.text)
```

这里我向欢迎页面发送了一个登录参数，它的作用就像登录表单的处理器。然后我从请求结果中获取 cookie，打印登录状态的验证结果，然后再通过 `cookies` 参数把 cookie 发送到简介页面。

对于简单的访问，这样处理没有问题，但是如果你面对的网站比较复杂，它经常暗自调整 cookie，或者如果你从一开始就完全不想用 cookie，该怎么处理呢？ Requests 库的 `session` 函数可以完美地解决这个问题：

```
import requests

session = requests.Session()

params = {'username': 'username', 'password': 'password'}
s = session.post('http://pythonscraping.com/pages/cookies/welcome.php', params)
print('Cookie is set to:')
print(s.cookies.get_dict())
print('Going to profile page...')
s = session.get('http://pythonscraping.com/pages/cookies/profile.php')
print(s.text)
```

在这个例子中，会话（session）对象（通过调用 `requests.Session()` 获取）会持续跟踪会话信息，包括 cookie、header，甚至是 HTTP 协议的信息，比如 HTTPAdapter（为 HTTP 和 HTTPS 的链接会话提供统一接口）。

Requests 是一个非常给力的库，程序员完全不用费脑子，也不用写代码，可能只逊色于 Selenium（第 11 章将会介绍）。虽然写网络爬虫的时候，你可能想放手让 Requests 库替自己做所有的事情，但是持续关注 cookie 的状态，掌握它们可以控制的范围是非常重要的。这样可以避免痛苦地调试和分析网站的异常行为，节省很多时间。

HTTP基本接入认证

在发明 cookie 之前，处理网站登录的一种常用方法就是用 **HTTP 基本接入认证**（HTTP basic access authentication）。你会时不时见到它们，尤其是在一些安全性较高的网站或公司网站上，以及一些 API 上。我在 http://pythonscraping.com/pages/auth/login.php 用这种认证方法创建了一个页面（见图 10-2）。

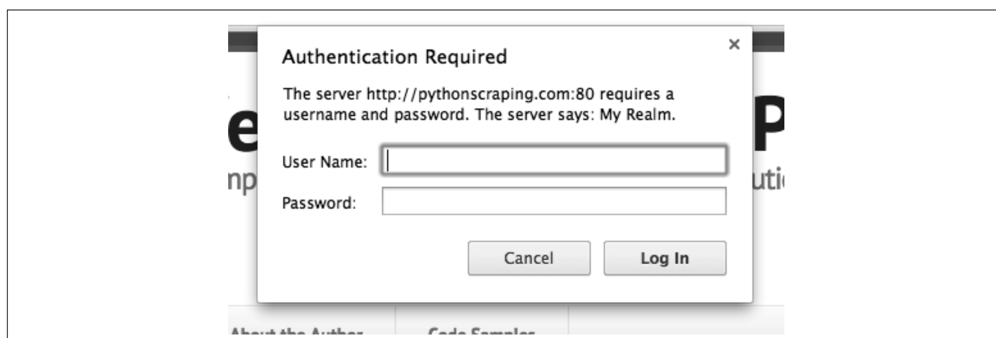


图 10-2：基本接入认证页面，用户必须提供用户名和密码才能登录

和前面的例子一样，你可以用任意用户名登录，但是密码必须是“password”。

Requests 库有一个 `auth` 模块，专门用来处理 HTTP 认证：

```
import requests
from requests.auth import AuthBase
from requests.auth import HTTPBasicAuth

auth = HTTPBasicAuth('ryan', 'password')
r = requests.post(url='http://pythonscraping.com/pages/auth/login.php', auth=
                  auth)
print(r.text)
```

虽然这看着像是一个普通的 `POST` 请求，但是有一个 `HTTPBasicAuth` 对象作为 `auth` 参数传递到了请求中。显示的结果将是用户名和密码验证成功的页面（如果验证失败，就是一个拒绝接入页面）。

10.6　其他表单问题

网页表单是网络恶意机器人（malicious bots）酷爱的网站切入点。你当然不希望机器人创建垃圾账号，占用昂贵的服务器资源，或者在博客上提交垃圾评论。因此，现代网站经常在 HTML 中采取很多安全措施，让表单不能被快速穿越。



关于验证码（CAPTCHA）的作用，请查看第 13 章内容，里面介绍了 Python 的图像处理和文本识别方法。

如果你在提交表单时遇到了一个莫名其妙的错误，或者服务器一直以陌生的理由拒绝你，请查看第 14 章内容，里面介绍了蜜罐（honey pot）、隐含字段（hidden field），以及其他保护网页表单的安全措施。

第 11 章

抓取JavaScript

客户端脚本语言是运行在浏览器而非服务器上的语言。客户端语言成功的前提是浏览器能够正确地解释和执行这类语言（这也是在浏览器上禁用 JavaScript 非常容易的原因）。

在一定程度上，由于很难让所有浏览器开发商都认可同一个标准，所以客户端语言比服务器端语言要少很多。不过这在网页抓取的时候是件好事：要处理的语言越少越好。

通常，你在网上遇到的客户端语言只有两种：ActionScript（开发 Flash 应用的语言）和 JavaScript。今天 ActionScript 的使用率比 10 年前低很多，它经常用于流媒体文件播放，用作在线游戏的平台，或者用于那些没人想看的网站“介绍”页面。总之，抓取 Flash 页面的需求并不多，所以本章重点介绍现代网页中普遍使用的客户端语言：JavaScript。

到目前为止，JavaScript 是 Web 上最常用也是支持者最多的客户端脚本语言。它可以收集用户跟踪数据，不需要重载页面直接提交表单，在页面中嵌入多媒体文件，甚至运行在线游戏。那些看起来非常简单的页面背后通常使用了许多 JavaScript 文件。你可以在网页源代码的 `<script>` 标签之间看到它们：

```
<script>
    alert("This creates a pop-up using JavaScript");
</script>
```

11.1 JavaScript简介

对要抓取的语言预先做些了解会很有用。自己熟悉一下 JavaScript 总会有好处。

JavaScript 是一种弱类型语言，其语法通常可以与 C++ 和 Java 相比。虽然语法中的一些元

素，比如操作符、循环条件和数组，都与 C++、Java 语法很接近，但是 JavaScript 的弱类型和脚本形式被一些程序员看成是折磨人的"怪兽"。

例如，下面的 JavaScript 程序通过递归方式计算斐波纳契序列，最后把结果打印在浏览器的开发者控制台里：

```
<script>
function fibonacci(a, b){
    var nextNum = a + b;
    console.log(nextNum+" is in the Fibonacci sequence");
    if(nextNum < 100){
        fibonacci(b, nextNum);
    }
}
fibonacci(1, 1);
</script>
```

注意，JavaScript 里所有的变量都用 `var` 关键字进行定义。这与 PHP 里的 `$` 符号，或者 Java 和 C++ 里的类型声明（`int`、`String`、`List` 等）类似。Python 不太一样，它没有这种显式的变量声明。

JavaScript 还有一个非常好的特性，就是把函数作为变量使用：

```
<script>
var fibonacci = function() {
    var a = 1;
    var b = 1;
    return function () {
        var temp = b;
        b = a + b;
        a = temp;
        return b;
    }
}
var fibInstance = fibonacci();
console.log(fibInstance()+" is in the Fibonacci sequence");
console.log(fibInstance()+" is in the Fibonacci sequence");
console.log(fibInstance()+" is in the Fibonacci sequence");
</script>
```

这段代码乍看起来可能有点儿恐怖，不过如果你把这种特性看成 Lambda 表达式（第 2 章介绍过），就很简单了。变量 `fibonacci` 被定义成一个函数。它的函数值返回一个函数，该函数会打印斐波纳契序列里不断增大的值。每次被调用时，它都会返回斐波纳契的计算函数，该函数再次执行序列计算，并增加函数变量的值。

虽然乍看起来有点儿复杂，但是在解决一些问题时，比如计算斐波纳契序列值，这种模式还是比较合适的。在处理用户行为和回调函数时，把函数作为变量进行传递是非常方便的，另外在阅读 JavaScript 代码的时候也有必要适应这种编程方式。

常用JavaScript库

虽然了解 JavaScript 语言本身的语法很重要，但是在现代 Web 上，你至少得使用一种 JavaScript 第三方库。在查看网页源代码的时候，你可能会看到一种或多种常用的 JavaScript 库。

用 Python 执行 JavaScript 代码的效率非常低，既费时又费力，尤其是在处理规模较大的 JavaScript 代码时。如果有绕过 JavaScript 并直接解析它的方法（不需要执行它就可以获得信息）会非常实用，可以帮你避开一大堆麻烦事。

1. jQuery

jQuery 是一个十分常见的库，70% 的最流行的网站（约 200 万）和约 30% 的其他网站（约 2 亿）都在使用。[1] 使用了 jQuery 的网站很好识别，其源代码里包含了 jQuery 的入口，比如：

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></
script>
```

如果你在一个网站上看到了 jQuery，那么抓取这个网站的数据时要格外小心。jQuery 可以动态地创建 HTML 内容，这些内容只有在 JavaScript 代码执行之后才会显示。如果你用传统的方法抓取页面内容，就只能获得 JavaScript 代码执行之前页面上的内容（11.2 节会详细介绍这个抓取问题）。

另外，这些页面很可能包含动画、用户交互内容和嵌入式媒体，这些内容都增加了网页抓取的难度。

2. Google Analytics

大约一半的网站都在用 Google Analytics[2]，它可能是网站最常用的 JavaScript 库和最受欢迎的用户跟踪工具。http://pythonscraping.com 和 http://www.oreilly.com/ 都用了 Google Analytics。

很容易判断一个页面是不是使用了 Google Analytics。如果网站使用了它，在页面底部会有类似如下所示的 JavaScript 代码（取自 O'Reilly Media 网站）：

```
<!-- Google Analytics -->
<script type="text/javascript">

var _gaq = _gaq || [];
_gaq.push(['_setAccount', 'UA-4591498-1']);
_gaq.push(['_setDomainName', 'oreilly.com']);
_gaq.push(['_addIgnoredRef', 'oreilly.com']);
_gaq.push(['_setSiteSpeedSampleRate', 50]);
```

注 1：Dave Methvin 于 2014 年 1 月 13 日在他的博客中发表了"The State of jQuery 2014"一文，里面包含了详细的统计数据。

注 2：W3Techs,"Usage Statistics and Market Share of Google Analytics for Websites".

```
_gaq.push(['_trackPageview']);

(function() { var ga = document.createElement('script'); ga.type =
'text/javascript'; ga.async = true; ga.src = ('https:' ==
document.location.protocol ? 'https://ssl' : 'http://www') +
'.google-analytics.com/ga.js'; var s =
document.getElementsByTagName('script')[0];
s.parentNode.insertBefore(ga, s); })();

</script>
```

这段代码处理用于跟踪页面访问的 Google Analytics 的 cookie。有时候，这对于设计用于执行 JavaScript 和处理 cookie 的爬虫（比如利用 Selenium 的那些，稍后讨论）来说会是个问题。

如果一个网站使用了 Google Analytics 或其他类似的网络分析系统，而你不想让网站知道你在抓取它的数据，就要确保把那些分析工具的 cookie 或者所有 cookie 都关掉。

3. Google Maps

只要你上过网，就一定见过内嵌 Google Maps 的网站。用 Google Maps 的 API 很容易在任何网站上嵌入带有自定义信息的地图。

如果你要抓取任何位置数据，理解 Google Maps 的工作方式可以让你轻松地获取格式规范的经纬度坐标和具体地址。在 Google Maps 上，显示一个位置最常用的一种方法就是用**标记**（一个大头针）。

可以用下面的代码将标记插在 Google Maps 上：

```
var marker = new google.maps.Marker({
      position: new google.maps.LatLng(-25.363882,131.044922),
      map: map,
      title: 'Some marker text'
  });
```

Python 可以轻松地抽取出所有位置在 `google.maps.LatLng()` 里的坐标，生成一组经纬度坐标值。

通过 Google 的 Reverse Geocoding API，你可以把这些经纬度坐标组解析成格式规范的地址，便于存储和分析。

11.2 Ajax和动态HTML

到目前为止，我们与 Web 服务器通信的唯一方式，就是发出 HTTP 请求获取新页面。如果提交表单之后，或者从 Web 服务器获取信息时，网站的页面不需要重新加载，那么你访问的网站很可能使用了 Ajax 技术。

与一些人的想法相反，Ajax 其实并不是一门语言，而是用来完成某个任务（细想一下，与

网页抓取差不多）的一系列技术。Ajax 的全称是 Asynchronous JavaScript and XML（异步 JavaScript 和 XML），网站不需要使用单独的页面请求就可以和 Web 服务器进行交互（收发信息）。



需要注意的是，你不应该说“这个网站是用 Ajax 写的”。正确的说法应该是“这个表单使用 Ajax 与 Web 服务器通信”。

和 Ajax 一样，**动态 HTML**（dynamic HTML，DHTML）也是用于某一常见目的的一系列技术。DHTML 是客户端脚本改变页面的 HTML 元素时，改变的 HTML 代码、CSS 语言，或者二者兼而有之。比如，按钮仅在用户移动光标之后才出现，背景色可能每次点击都会改变，或者用一个 Ajax 请求触发页面加载一段新内容。

值得注意的是，虽然“动态”这个词往往和“移动”或“变化”联系在一起，但是那些使用了交互式 HTML 组件、图像可以移动，或者带有嵌入式媒体文件的网页，并不一定就是 DHTML，即使页面看起来是动态的。另外，一些看起来极其单调、静态的网页，底层却可能是用 DHTML 处理的，关键要看有没有用 JavaScript 控制 HTML 和 CSS 元素。

如果你抓取过许多网站，很可能会遇到这样一种情况：你在浏览器上看到的内容，与你用爬虫从网站上抓取的内容不一样。你可能会怀疑自己是不是哪个细节没处理好，希望找出内容抓取不出来的原因。

有时你还会发现，网页用一个加载页面把你重定向到另一个结果页面，但是网页的 URL 链接在这个过程中一直没有变化。

这些都是因为你的爬虫不能执行那些让页面产生各种神奇效果的 JavaScript 代码。如果网站的 HTML 页面没有执行 JavaScript，就可能和你在浏览器里看到的样子完全不同，因为浏览器可以正确地执行 JavaScript。

对于那些使用了 Ajax 或 DHTML 技术来改变 / 加载内容的页面，可能有一些抓取手段，但是用 Python 解决这个问题只有两种途径：直接从 JavaScript 代码里抓取内容，或者用 Python 的第三方库执行 JavaScript，直接抓取你在浏览器里看到的页面。

11.2.1　在Python中用Selenium执行JavaScript

Selenium 是一个强大的网页抓取工具，最初是为网站自动化测试而开发的。近几年，它还被广泛用于获取精确的网站快照，因为网站可以直接运行在浏览器中。Selenium 可以让浏览器自动加载网站，获取需要的数据，甚至对页面截屏，或者判断网站上是否发生了某些操作。

Selenium 自己不带浏览器，它需要与第三方浏览器集成才能运行。例如，如果你在 Firefox 上运行 Selenium，会看到一个 Firefox 窗口被打开，进入网站，然后执行你在代码中设置

的动作。虽然这样可以看得更清楚，但是我更喜欢让程序在后台静静地运行，所以我用一个叫 PhantomJS 的工具代替真实的浏览器。

PhantomJS 是一个**无头浏览器**（headless browser）。它会把网站加载到内存并执行页面上的 JavaScript，但是它不会向用户展示网页的图形界面。把 Selenium 和 PhantomJS 结合在一起，就可以运行一个非常强大的网络爬虫来轻松处理 cookie、JavaScript、header 以及任何你需要做的事情。

你可以从 PyPI 网站下载 Selenium 库，也可以用第三方管理器（比如 pip）用命令行安装。

PhantomJS 可以从它的官方网站下载。因为 PhantomJS 是一个功能完善（虽然无头）的浏览器，并非一个 Python 库，所以需要下载并安装才能使用，并且不能用 pip 进行安装。

虽然很多页面都用 Ajax 加载数据（尤其是 Google），我还是在 http://pythonscraping.com/pages/javascript/ajaxDemo.html 创建了一个简单的页面来运行我们的爬虫。这个页面上有一些简单的文字，是手工敲在 HTML 代码里的，打开页面两秒钟之后，它们就会被替换成由 Ajax 生成的内容。如果我们用传统的方法抓取这个页面，只能获取加载页面，而我们真正需要的信息（Ajax 执行之后的页面）却抓不到。

Selenium 库是一个在 WebDriver 对象上调用的 API。WebDriver 有点儿像可以加载网站的浏览器，但是它也可以像 `BeautifulSoup` 对象一样用来查找页面元素，与页面上的元素交互（发送文本、点击等），以及执行其他动作来运行网络爬虫。

下面的代码可以获取前面测试页面上 Ajax“墙”后面的内容：

```
from selenium import webdriver
import time

driver = webdriver.PhantomJS(executable_path='<PhantomJS Path Here>')
driver.get('http://pythonscraping.com/pages/javascript/ajaxDemo.html')
time.sleep(3)
print(driver.find_element_by_id('content').text)
driver.close()
```

Selenium 的选择器

在之前的几章里，我们用过 BeautifulSoup 的选择器来选择页面的元素，比如 `find` 和 `findAll`。Selenium 在 WebDriver 的 DOM 中使用了一组全新的选择器来查找元素，不过它们都使用非常直截了当的名称。

在这个例子中，我们用的选择器是 `find_element_by_id`，但下面的其他选择器也可以获得同样的结果。

```
driver.find_element_by_css_selector('#content')
driver.find_element_by_tag_name('div')
```

当然，如果你想选择页面上的多个元素，大部分选择器都可以用 elements（复数）来返回一个 Python 列表：

```
driver.find_elements_by_css_selector('#content')
driver.find_elements_by_css_selector('div')
```

另外，如果你还是想用 BeautifulSoup 来解析网页内容，可以用 WebDriver 的 page_source 函数返回页面的源代码字符串。

```
pageSource = driver.page_source
bs = BeautifulSoup(pageSource, 'html.parser')
print(bs.find(id='content').get_text())
```

这段代码用 PhantomJS 库创建了一个新的 Selenium WebDriver，首先用 WebDriver 加载页面，然后暂停执行 3 秒钟，再查看页面以获取（希望已经加载完成的）内容。

依据你的 PhantomJS 安装位置，在创建新的 PhantomJS WebDriver 时，你可能还需要在 Selenium 的 WebDriver 接入点指明 PhantomJS 可执行文件的路径：

```
driver = webdriver.PhantomJS(executable_path='path/to/driver/'\
                             'phantomjs-1.9.8-macosx/bin/phantomjs')
```

如果一切配置正确，上面的程序会在几秒钟后显示下面的结果：

```
Here is some important text you want to retrieve!
A button to click!
```

需要注意的是，虽然页面里有一个元素是 HTML 按钮，但是 Selenium 的 .text 函数可以获取按钮的文本内容，就像获取页面上其他元素的内容一样。

如果 time.sleep 的暂停时间由 3 秒改成 1 秒，那么上面程序抓取的文本就会变成：

```
This is some content that will appear on the page while it's loading.
You don't care about scraping this.
```

虽然这个方法奏效了，但是效率不够高，在处理规模较大的网站时可能会出问题。页面的加载时间是不确定的，具体依赖于服务器某一毫秒的负载情况，以及不断变化的网速。虽然这个页面只需要 2 秒多的加载时间，但是我们设置了 3 秒的等待时间以确保页面完全加载。一种更加高效的方法是，让 Selenium 不断地检查某个元素是否存在，以此确定页面是否已经完全加载，如果页面加载成功就执行后面的程序。

下面的程序检查 ID 为 loadedButton 的按钮是否存在，以此判断页面是不是已经完全加载：

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
```

```
from selenium.webdriver.support import expected_conditions as EC

driver = webdriver.PhantomJS(executable_path='')
driver.get('http://pythonscraping.com/pages/javascript/ajaxDemo.html')
try:
    element = WebDriverWait(driver, 10).until(
                        EC.presence_of_element_located((By.ID, 'loadedButton')))
finally:
    print(driver.find_element_by_id('content').text)
    driver.close()
```

程序里导入了一些新的模块，最值得注意的是 `WebDriverWait` 和 `expected_conditions`，这两个模块组合起来构成了 Selenium 的**隐式等待**（implicit wait）。

隐式等待与显式等待的不同之处在于，隐式等待是等 DOM 中某个状态发生后再继续运行代码[3]，而显式等待明确设置了等待时间，如前面例子中的 3 秒钟。在隐式等待中，DOM 触发的状态是用 `expected_conditions` 定义的（这里导入后用了别名 `EC`，是常用的简称）。在 Selenium 库里面元素被触发的期望条件（expected condition）有很多种，包括：

- 弹出一个提示框
- 一个元素（比如文本框）被选中
- 页面的标题改变了，或者文本显示在页面上或者某个元素里
- 一个元素对 DOM 可见，或者一个元素从 DOM 中消失了

当然，大多数期望条件在使用前都需要你先指定等待的目标元素。元素用**定位器**（locator）指定。注意，定位器与选择器是不一样的（关于选择器的更多介绍，请参见前文的"Selenium 的选择器"）。定位器是一种抽象的查询语言，用 `By` 对象表示，可以用于不同的场合，包括创建选择器。

在下面的示例代码中，一个定位器被用来查找 ID 为 `loadedButton` 的按钮：

```
EC.presence_of_element_located((By.ID, 'loadedButton'))
```

定位器还可以用来创建选择器，配合 WebDriver 的 `find_element` 函数使用：

```
print(driver.find_element(By.ID, 'content').text)
```

下面这行代码的功能和示例代码中一样：

```
print(driver.find_element_by_id('content').text)
```

如果你可以不用定位器，就不要用，毕竟这样可以少导入一个模块。但是，定位器是一种十分方便的工具，可以用在不同的应用中，并且具有很强的灵活性。

注 3：没有明确的等待时间，但是有最大等待时限，只要在时限内就可以。——译者注

下面是定位器通过 `By` 对象进行选择的策略。

`ID`

在上面的例子里用过；通过 HTML 的 `id` 属性查找元素。

`CLASS_NAME`

通过 HTML 的 `class` 属性来查找元素。为什么这个函数是 `CLASS_NAME`，而不是简单的 `CLASS`？在 Selenium 的 Java 库里使用 `object.CLASS` 可能会出现问题，`.class` 是 Java 保留的一个方法。为了让 Selenium 语法可以兼容不同的语言，就用 `CLASS_NAME` 代替。

`CSS_SELECTOR`

通过 CSS 的 `class`、`id`、`tag` 属性名来查找元素，用 `#idName`、`.className`、`tagName` 表示。

`LINK_TEXT`

通过链接文字查找 HTML 的 `<a>` 标签。例如，如果一个链接的文字是“Next”，就可以用 `(By.LINK_TEXT, "Next")` 来选择。

`PARTIAL_LINK_TEXT`

与 `LINK_TEXT` 类似，只是通过部分链接文字来查找。

`NAME`

通过 `name` 属性查找 HTML 标签。这在处理 HTML 表单时非常方便。

`TAG_NAME`

通过标签的名称查找 HTML 标签。

`XPATH`

用 `XPath` 表达式选择匹配的元素。

XPath 语法

XPath（XML Path，XML 路径）是在 XML 文档中导航和选择元素的查询语言。它由 W3C 于 1999 年创建，在 Python、Java 和 C# 这些语言中有时被用来处理 XML 文档。

虽然 BeautifulSoup 不支持 XPath，但是本书中的很多库（例如 Selenium 和 Scrapy）都支持。它的使用方式通常和 CSS 选择器（比如 `mytag#idname`）一样，虽然它原本被设计用于处理更规范的 XML 文档而不是 HTML 文档。

XPath 语法中有 4 个重要概念。

- 根节点和非根节点
  - `/div` 选择 div 节点，只有当它是文档的根节点时
  - `//div` 选择文档中所有的 div 节点（包括非根节点）

- 通过属性选择节点
  - `//@href` 选择带 `href` 属性的所有节点
  - `//a[@href='http://google.com']` 选择文档中所有指向 Google 网站的链接
- 通过位置选择节点
  - `//a[3]` 选择文档中的第 3 个链接
  - `//table[last()]` 选择文档中的最后一个表
  - `//a[position() < 3]` 选择文档中的前 3 个链接
- 星号（*）匹配任意字符或节点，可以在不同情况下使用
  - `//table/tr/*` 选择所有表格中 `tr` 标签的所有子节点（这很适合选择 `th` 和 `td` 标签）
  - `//div[@*]` 选择带任意属性的所有 `div` 标签

当然，XPath 还有很多高级的语法特征。经过这些年的发展，它已经变成一种非常复杂的查询语言，可以使用布尔逻辑、函数（如 `position()`），以及大量这里没介绍的操作符。

如果这里介绍的几个 XPath 功能解决不了你的 HTML 或 XML 元素选择问题，请参考微软的 XPath 语法页面。

11.2.2　Selenium的其他webdriver

在前一节中，Selenium 使用的是 PhantomJS driver。在大多数情况下，浏览器都不会弹出，就可以直接开始抓取网站，因此像 PhantomJS 这样的无头浏览器是非常方便的。但是，基于以下几个原因，使用另外一种类型的浏览器来运行你的爬虫可能很有用。

- 故障排除。如果你的代码运行的是 PhantomJS 并且运行失败了，那么在页面并未展示在你眼前的情况下，很难对错误进行诊断。如果使用其他的浏览器，你可以在任意断点暂停代码的运行，并与网页进行交互。
- 测试可能只能采用特定的浏览器来运行。
- 非常注重细节的网站在不同浏览器上的表现可能不同。你的代码可能在 PhantomJS 中不奏效。

很多官方和非官方的小组都在为当今主流的浏览器创建和维护 Selenium webdriver。Selenium 小组创建了一个 webdriver 集合（http://www.seleniumhq.org/download/）以供参考。

```
firefox_driver = webdriver.Firefox('<path to Firefox webdriver>')
chrome_driver = webdriver.Chrome('<path to Chrome webdriver>')
safari_driver = webdriver.Safari('<path to Safari webdriver>')
ie_driver = webdriver.Ie('<path to Internet Explorer webdriver>')
```

11.3 处理重定向

客户端重定向是在服务器将页面内容发送到浏览器之前，由浏览器执行 JavaScript 完成的页面跳转，而不是服务器完成的跳转。当使用浏览器访问页面的时候，有时很难区分这两种重定向。由于客户端重定向执行得很快，加载页面时你甚至感觉不到任何延迟，所以会让你觉得这个重定向就是一个服务器端重定向。

但是，在进行网页抓取的时候，这两种重定向的差异是非常明显的。根据处理方式，服务器端重定向可以轻松地通过 Python 的 `urllib` 库解决，而不需要使用 Selenium（更多信息请参考第 3 章）。客户端重定向却不能这样处理，除非你有工具可以执行 JavaScript。

Selenium 可以执行这种 JavaScript 重定向，就像执行其他 JavaScript 一样；但是这类重定向的主要问题是什么时候停止页面执行，也就是说，怎么判断一个页面已经完成重定向。在 http://pythonscraping.com/pages/javascript/redirectDemo1.html 的示例页面是客户端重定向的一个例子，有 2 秒的延迟。

我们可以用一种智能的方法来检测客户端重定向是否已完成。首先从页面开始加载时就“监视”DOM 中的一个元素，然后重复调用这个元素，直到 Selenium 抛出一个 `StaleElementReferenceException` 异常；也就是说，元素不在页面的 DOM 里了，这说明此时网站已经跳转。

```
from selenium import webdriver
import time
from selenium.webdriver.remote.webelement import WebElement
from selenium.common.exceptions import StaleElementReferenceException

def waitForLoad(driver):
    elem = driver.find_element_by_tag_name("html")
    count = 0
    while True:
        count += 1
        if count > 20:
            print('Timing out after 10 seconds and returning')
            return
        time.sleep(.5)
        try:
            elem == driver.find_element_by_tag_name('html')
        except StaleElementReferenceException:
            return

driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get('http://pythonscraping.com/pages/javascript/redirectDemo1.html')
waitForLoad(driver)
print(driver.page_source)
```

这个程序每半秒钟检查一次网页，看看 `html` 标签还在不在，时限为 10 秒钟，不过检查的时间间隔和时限都可以按需随意调整。

另外，你可以编写一个类似的循环来检查当前页面的 URL，直到 URL 发生改变，或者匹配到你寻找的特定的 URL。

等待元素的出现和消失是 Selenium 中一个常见的任务，你也可以使用前面按钮加载示例中的 `WebDriverWait` 函数。这里提供一个 15 秒钟的时限和一个 XPath 选择器，该 XPath 选择器寻找页面内容以完成同样的任务。

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import TimeoutException

driver = webdriver.PhantomJS(executable_path=
    'drivers/phantomjs/phantomjs-2.1.1-macosx/bin/phantomjs')
driver.get('http://pythonscraping.com/pages/javascript/redirectDemo1.html')
try:
    bodyElement = WebDriverWait(driver, 15).until(EC.presence_of_element_located(
        (By.XPATH, '//body[contains(text(),
        "This is the page you are looking for!)]")))
    print(bodyElement.text)
except TimeoutException:
    print('Did not find the element')
```

11.4　关于JavaScript的最后提醒

如今，大多数网站都使用了 JavaScript。[4] 幸运的是，在大多数情况下，JavaScript 的使用并不影响你抓取网页的方式。JavaScript 可能仅用来控制网站的跟踪工具、控制网站的一小部分，或者操作一个下拉菜单。如果 JavaScript 确实影响你抓取网站的方式，也很容易通过 Selenium 这样的工具来执行它，以生成简单的 HTML 页面（你已经在本书的第一部分学会了如何抓取）。

请记住：一个网站使用 JavaScript 并不意味着所有传统的网页抓取工具都失效了。JavaScript 的目标是生成 HTML 和 CSS，然后被浏览器渲染，或者是通过 HTTP 请求和响应与服务器动态通信。一旦使用了 Selenium，页面上的 HTML 和 CSS 就可以和其他网站代码一样被读取和解析，HTTP 请求和响应也可以在不使用 Selenium 的情况下用前几章中介绍的技术来发送和处理。

另外，JavaScript 对于网络爬虫来说甚至会带来一些好处，因为它作为“浏览器端的内容管理系统”，可能会向外界暴露出有用的 API，让你可以更直接地获取数据。更多的相关信息，请参见第 12 章。

如果你仍然难以应对某种复杂的 JavaScript 情形，可以到第 15 章查找关于 Selenium 以及直接与动态网站交互（包括拖放动作）的信息。

注 4：W3Techs,“Usage of JavaScript for Websites”.

第12章

利用API抓取数据

JavaScript 曾经是网络爬虫的灾难。在互联网的悠久历史中，你通过向 Web 服务器发起请求所获取到的数据，一度与用户在发起同样的请求后在 Web 浏览器中看到的数据是一样的。

随着 JavaScript 和 Ajax 内容的生成和加载变得越来越普遍，这种情况变得越来越少见了。在第 11 章，你看到了解决该问题的方法之一：利用 Selenium 让浏览器自动加载网站并获取数据。这是一种简便的方法，并且在大多数时候都是有效的。

问题是，当你有一个像 Selenium 一样强大和有效的“利器”时，每一个网页抓取问题都是小菜一碟。

在这一章里，你将完全不用理会 JavaScript（没有必要运行甚至是加载 JavaScript），直接获得数据源：生成数据的 API。

12.1 API概述

尽管关于 REST、GraphQL、JSON 和 XML API 的图书、演讲和指南不胜枚举，但它们都是基于一个简单的概念。API 定义了允许一个软件与另一个软件通信的标准语法，即便是这两个软件是用不同的语言编写的或者是架构不同。

本节重点介绍 Web API（特别是允许 Web 服务器与浏览器交流的 API），并用 API 这个词特指这一类 API。但是你也需要注意，在其他上下文中，API 也会被用作一个通用的词，指允许 Java 程序与 Python 程序在同一台机器上通信的接口。API 并不一定是“跨网络的”，也并不总是涉及任何 Web 技术。

Web API 经常被那些使用成熟的公开服务（public service）的开发者所使用。例如，ESPN 提供了获取运动员信息、比赛分数等信息的 API（http://www.espn.com/apis/devcenter/docs/）。Google 的开发者社区也提供了几十个 API，用于获取语言翻译、分析、地理位置等信息。

这些 API 的文档通常将路由或者**端点**（endpoint）描述为你可以请求的 URL，而变量参数要么是 URL 路径，要么是 `GET` 请求的参数。

例如，以下示例提供了 `pathparam` 作为路由路径里的一个参数：

```
http://example.com/the-api-route/pathparam
```

而以下示例则是将 `pathparam` 作为 param1 的参数值：

```
http://example.com/the-api-route?param1=pathparam
```

以上两种传递变量数据给 API 的方法都很常用，尽管像很多计算机科学领域的许多主题一样，关于应该在何时以及在哪里通过路径或者参数来传递这些变量，也曾经发生过激烈的哲学式辩论。

API 的响应通常是 JSON 或者 XML 格式的。现在 JSON 远比 XML 流行，但是你仍然能看到一些 XML 响应。很多 API 允许你改变响应类型，通常用另外一个参数定义你希望的响应类型。

以下是 JSON 格式的 API 响应示例：

```
{"user":{"id": 123, "name": "Ryan Mitchell", "city": "Boston"}}
```

以下是 XML 格式的 API 响应示例：

```
<user><id>123</id><name>Ryan Mitchell</name><city>Boston</city></user>
```

淘宝 IP 地址库（http://ip.taobao.com）提供了一个简单易用的 API，它能将 IP 地址翻译成实际的物理地址。你可以在浏览器中输入下面的网址，发起一个简单的 API 请求：[1]

```
http://ip.taobao.com/service/getIpInfo.php?ip=50.78.253.58
```

这应该会生成下面的结果：

```
{"code":0,"data":{"ip":"50.78.253.58","country":"美国","area":"","region":
 "康涅狄格","city":"哈特福德","county":"XX","isp":"康卡斯特","country_id":
 "US","area_id":"","region_id":"US_107","city_id":"US_1049","county_id":
 "xx","isp_id":"30007"}}
```

注 1：这个 API 把 IP 地址解析成地理位置，本章后面还会用到这个 API。

12.1.1 HTTP方法和API

前面你看到了如何利用 API 向服务器发送 GET 以获取信息。利用 HTTP 从 Web 服务器获取信息有 4 种方式（或**方法**）：

- GET
- POST
- PUT
- DELETE

从技术上看，不止存在以上 4 种方式（例如 HEAD、OPTIONS 和 CONNECT），但是它们在 API 中很少会用到，你也不可能会碰到这些方式。大多数 API 仅提供了以上 4 种方法，甚至是这 4 种方法的一个子集。所以 API 仅仅使用 GET 方法，或者仅仅使用 GET 和 POST 方法是很常见的。

你在浏览器地址栏中输入网址访问网站时，使用的就是 GET。当你访问 http://ip.taobao.com/service/getIpInfo.php?ip=50.78.253.58 时，就会使用 GET 方法。可以想象成 GET 在说："喂，Web 服务器，请按照这个网址为我提供信息。"

从定义上看，GET 请求对服务器数据库的信息不会有任何影响。它不会存储任何信息，也不会修改任何信息，只是读取信息。

当你填写表单或提交信息到 Web 服务器的后端程序时，使用的就是 POST。每次当你登录一个网站的时候，就是通过用户名和（希望是）加密的密码发起一个 POST 请求。如果你用 API 发起一个 POST 请求，相当于说"请把这个信息保存到你的数据库里"。

PUT 在与网站交互时不常用，但是在 API 里面有时会用到。PUT 请求用来更新一个对象或信息。例如，API 可能会要求用 POST 请求创建新用户，但是如果你要更新老用户的邮箱地址，就要用 PUT 请求了。[2]

DELETE 用于删除对象。例如，如果我们向 http://myapi.com/user/23 发出一个 DELETE 请求，就会删除 ID 号是 23 的用户。DELETE 方法在公共 API 里面不常用，公共 API 主要用于传播信息或者允许用户创建或发布信息，而不是让用户删掉数据库中的信息。

与 GET 请求不同，除了你请求数据的 URL 或路由以外，POST、PUT 和 DELETE 请求还允许你在请求体中发送其他信息。

注 2：其实，很多 API 在更新信息的时候都是用 POST 请求代替 PUT 请求。究竟是创建一个新实体还是更新一个旧实体，通常要看 API 请求本身是如何构建的。不过，掌握两者的差异还是有好处的，在常用的 API 中你经常会遇到 PUT 请求。

和你从 Web 服务器接收到的响应一样，请求体中的这个数据通常也是 JSON 格式的，有时是 XML 格式的，而且数据的格式是在 API 的语法中定义好的。例如，如果你用一个 API 创建博客文章的评论，可能会发送一个 PUT 请求到：

```
http://example.com/comments?post=123
```

请求体如下：

```
{"title": "Great post about APIs!", "body": "Very informative. Really helped me
out with a tricky technical challenge I was facing. Thanks for taking the time
to write such a detailed blog post about PUT requests!", "author": {"name": "Ryan
Mitchell", "website": "http://pythonscraping.com", "company": "O'Reilly Media"}}
```

注意，这里博客文章的 ID（123）作为参数传入 URL，即你做出的新评论的内容通过请求体传送。参数和数据可以在参数变量和请求体中同时传送。而需要哪些参数以及在哪里传送依然是由 API 的语法决定的。

12.1.2　更多关于API响应的介绍

如你在本章开头的淘宝 IP 地址库示例中所见，API 的一个重要特性是会返回格式良好的响应。最常见的响应格式是 XML（eXtensible Markup Language，可扩展标记语言）和 JSON（JavaScript Object Notation，JavaScript 对象表示法）。

这几年，JSON 比 XML 受欢迎得多，主要有两个原因。首先，JSON 文件通常比设计良好的 XML 文件小。比如下面的 XML 数据用了 98 个字符：

```
<user><firstname>Ryan</firstname><lastname>Mitchell</lastname><username>Kludgist
</username></user>
```

同样的 JSON 格式的数据只需 73 个字符，比表述同样内容的 XML 文件要小 36%：

```
{"user":{"firstname":"Ryan","lastname":"Mitchell","username":"Kludgist"}}
```

当然，有人可能会说，XML 也可以表示成这种形式：

```
<user firstname="ryan" lastname="mitchell" username="Kludgist"></user>
```

不过这么做并不好，因为它不支持深层嵌入数据。而且它仍然需要 71 个字符，和 JSON 差不多。

JSON 格式比 XML 更受欢迎的另一个原因是 Web 技术的改变。过去，服务器端用 PHP 和 .NET 这些程序作为 API 的接收端。现在，服务器端也会用一些 JavaScript 框架作为 API 的发送和接收端，比如 Angular 或 Backbone 等。虽然服务器端的技术无法预测它们即将收到的数据格式，但是像 Backbone 之类的 JavaScript 库处理 JSON 要比处理 XML 简单。

尽管通常认为 API 的响应要么是 XML 格式要么是 JSON 格式，但是其他任何格式都是可能的。API 的响应类型受限于创建它的程序员的想象力。CSV 是另外一种典型的响应输出。一些 API 甚至被设计用来生成文件输出。一个请求可能是要求服务器生成一幅带有特定文本的图像，或者请求特定的 XLSX 或 PDF 文件。

一些 API 完全没有响应。例如，如果你向服务器请求生成一个新的博文评论，它可能仅返回一个 HTTP 响应代码 200，意思是："我发布了评论，一切都很好！"其他 API 可能返回一个如下所示的最小响应：

```
{"success": true}
```

如果发生了错误，你可能会得到如下响应：

```
{"error": {"message": "Something super bad happened"}}
```

如果 API 没有很好地进行配置，你得到的可能是一个不可解析的栈跟踪（stack trace）或者一些普通的英文文本。当向 API 发出一个请求时，明智的做法通常是首先检查你得到的响应是 JSON 格式（或者是 XML、CSV 或其他你期望的格式）。

12.2 解析JSON数据

在本章中，我们介绍了许多不同类型的 API 以及它们的使用方法，也介绍了这些 API 返回的 JSON 格式的响应。现在让我们看看如何解析并使用这些信息。

本章开始的时候，我举过淘宝 IP 地址库网站查 IP 的例子，它可以把 IP 地址解析转换成地理位置：

```
http://ip.taobao.com/service/getIpInfo.php?ip=50.78.253.58
```

我可以获取这个请求的响应输出，然后用 Python 的 JSON 解析函数来解码：

```
import json
fromurllib.requestimporturlopen

defgetCountry(ipAddress):
    response = urlopen("http://ip.taobao.com/service/getIpInfo.php?ip="
        +ipAddress).read().decode('utf-8')
responseJson=json.loads(response)
returnresponseJson.get("data")["country"]

print(getCountry("50.78.253.58"))
```

这段代码可以打印出 IP 地址为 50.78.253.58 的国家名称。

这里用的 JSON 解析库是 Python 标准库的一部分。只需在代码开头写上 `import json`，你就可以使用它了！不同于那些先将 JSON 解析成一种特殊的 JSON 对象或 JSON 节点的语言，Python 使用了一种更加灵活的方式，将 JSON 对象转换成字典，将 JSON 数组转换成列表，将 JSON 字符串转换成 Python 字符串，等等。通过这种方式，获取和操作 JSON 中存储的值就变得非常简单了。

下面的例子演示了如何使用 Python 的 JSON 解析库，处理 JSON 字符串中可能出现的不同数据类型：

```
import json

jsonString = '{"arrayOfNums":[{"number":0},{"number":1},{"number":2}],
               "arrayOfFruits":[{"fruit":"apple"},{"fruit":"banana"},
                                {"fruit":"pear"}]}'
jsonObj = json.loads(jsonString)

print(jsonObj.get('arrayOfNums'))
print(jsonObj.get('arrayOfNums')[1])
print(jsonObj.get('arrayOfNums')[1].get('number') +
      jsonObj.get('arrayOfNums')[2].get('number'))
print(jsonObj.get('arrayOfFruits')[2].get('fruit'))
```

输出的结果是：

```
[{'number': 0}, {'number': 1}, {'number': 2}]
{'number': 1}
3
pear
```

第一行是一个词典对象列表，第二行是一个词典对象，第三行是一个整数（第一行词典列表中整数的和），第四行是一个字符串。

12.3 无文档的API

到目前为止，本章只讨论了有文档的 API。它们的开发者希望它们被公众所使用，并发布了关于 API 的信息，并且假定这些 API 会被其他开发者使用。但是大多数 API 是没有发布任何文档的。

但是为什么你会创建一个 API 而不提供公开文档呢？正如本章开头提到的，这一切都与 JavaScript 有关。

通常，在用户请求一个网页时，动态网站的 Web 服务器会做以下几件事情：

- 处理来自请求网站页面的用户的 `GET` 请求
- 从数据库检索页面需要呈现的数据
- 按照 HTML 模板组织页面数据

- 发送带格式的 HTML 给用户

由于 JavaScript 框架变得越来越普遍，很多 HTML 创建任务从原来的由服务器处理变成了由浏览器处理。服务器可能给用户浏览器发送一个硬编码的 HTML 模板，但是还需要单独的 Ajax 请求来加载内容，并将这些内容放到 HTML 模板中正确的位置（slot）。所有这些都发生在浏览器 / 客户端上。

最初，上述机制对于网络爬虫来说是一个麻烦的问题。过去，爬虫请求一个 HTML 页面时，获取到的就是原封不动的 HTML 页面，所有的内容都在 HTML 页面上。而现在，爬虫获得的是一个不带有任何内容的 HTML 模板。

Selenium 就是用来解决这个问题的。现在，程序员的网络爬虫可以变成浏览器，请求 HTML 模板，执行任意的 JavaScript，允许加载所有的数据，然后再抓取网页的数据。由于 HTML 都被加载了，现在基本上只剩下之前已经解决了的问题——解析和格式化已有 HTML 的问题。

然而，由于整个内容管理系统（曾经只位于 Web 服务器中）基本上已经移到了浏览器端，连最简单的网站都可以激增至几兆字节的内容和十几个 HTTP 请求。

此外，当使用 Selenium 时，用户不需要的“额外信息”也被加载了。调用跟踪程序、加载侧边栏广告、调用侧边栏广告的跟踪程序。图像、CSS、第三方的字体数据——所有这些数据都被加载了。当你使用浏览器浏览网站的时候，这些内容可能看起来很好，但是当你编写一个需要快速移动、抓取特定数据并尽可能对 Web 服务器造成较小负担的爬虫的时候，这可能会加载比你实际所需多上百倍的数据。

但是对于 JavaScript、Ajax 和现代化 Web 来说仍有一线希望：因为服务器不再将数据处理成 HTML 格式，所以它们通常作为数据库本身的一个弱包装器。该弱包装器简单地从数据库中抽取数据，并通过一个 API 将数据返回给页面。

当然，这些 API 并未打算供除网页本身以外的任何人或者任何事使用，因此开发者未为这些 API 提供文档，并且假设（或者说希望）没有人会发现这些 API。但是这些 API 的确是存在的。

例如，《纽约时报》网站通过 JSON 加载所有的搜索结果。如果你访问以下链接：

```
https://query.nytimes.com/search/sitesearch/#/python
```

它呈现的是关于搜索词“python”的最新文章。如果你用 urllib 或 Request 库抓取这个页面，不会找到任何搜索结果。这些结果是通过一个 API 调用单独加载的：

```
https://query.nytimes.com/svc/add/v1/sitesearch.json
?q=python&spotlight=true&facet=true
```

如果你用 Selenium 加载该页面，将发起 100 次请求并在每次搜索时传输 600~700KB 的数据。直接使用 API 的话，你只需发起一次请求，并且只传输你所需要的大约 60KB 的格式良好的数据。

12.3.1 查找无文档的API

你在前面的章节使用 Chrome 检查器查看过 HTML 页面的内容，现在你要用它来实现不同的目的：查看用于构建页面的调用的请求和响应。

为此，打开 Chrome 检查器窗口并点击网络选项卡，如图 12-1 所示。

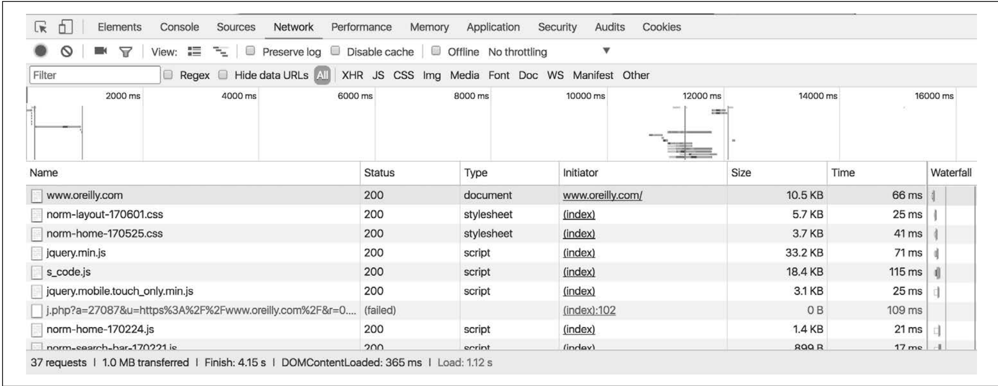


图 12-1：Chrome 网络检查器工具展示了浏览器所发起的和接收的所有调用

注意，你需要在页面加载前就打开这个窗口。当关闭时，它不会追踪网络调用。

当页面正在加载时，每当浏览器接收到 Web 服务器返回的页面渲染信息时，你将会实时看到一条线。这可能也包括一次 API 调用。

查找无文档的 API 需要做一些侦查工作（若不想做此侦查工作，请查看 12.3.3 节），特别是有很多网络调用的大型网站。通常来说，你看到它就会知道它是无文档的 API。

API 调用有几个特征，这些特征对于在网络调用列表中找到它们非常有用。

- 它们通常包含 JSON 或 XML。你可以利用搜索 / 过滤字段过滤请求列表。
- 利用 `GET` 请求，URL 中会包含一个传递给它们的参数。如果你要寻找一个返回搜索结果或者加载特定页面数据的 API 调用，这将非常有用。只需用你使用的搜索词、页面 ID 或者其他的识别信息，过滤结果即可。
- 它们通常是 XHR 类型的。

API 可能并不总是很明显，特别是在带有很多特征（在加载单个页面时可能会调用上百次）的大型网站中。但是通过一些练习，在干草堆中发现一根针会容易得多。

12.3.2 记录未被记录的API

在你发现一次 API 调用后，在某种程度上来说将其记录下来是非常有用的，你的爬虫严重依赖于这个调用时尤其如此。你可能需要在网站上加载多个页面，在检查器控制台的网络选项卡中筛选出目标 API 调用。这样做之后，你可以看到这个调用在不同页面的变化，并且识别出该调用接收的字段和返回的字段。

每个 API 调用都可以通过留心以下几个字段识别和记录下来：

- 使用的 HTTP 方法
- 输入
  - 路径参数
  - 请求头（包括 cookie）
  - 正文内容（对于 `PUT` 和 `POST` 调用）
- 输出
  - 响应头（包括 cookie 集合）
  - 响应正文类型
  - 响应正文字段

12.3.3 自动查找和记录API

查找和记录 API 看起来是一项烦琐和偏算法的工作。确实是这样。而且有些网站可能尝试混淆浏览器是如何获得数据的，这就使得这个任务变得更加困难，查找和记录 API 主要是一个程序性任务。

我在 https://github.com/REMitchell/apiscraper 创建了一个 GitHub 仓库，试图帮助完成其中部分烦琐的工作。

该工具会使用 Selenium、ChromeDriver 和一个叫作 BrowserMob Proxy 的库来加载页面，在一个域内抓取网页，分析页面加载过程中的网络流量，并将这些请求组织成可读的 API 调用。

为了让这个项目运转起来需要几个部件。首先就是该软件本身。

克隆 GitHub 项目 apiscraper。克隆项目应该包含以下文件。

apicall.py

  它包含定义一个 API 调用的属性（路径、参数等），以及确定两个 API 调用是否相同的逻辑。

apiFinder.py

  它是一个主抓取类。被 webservice.py 和 consoleservice.py 用来实现查找 API 的过程。

browser.py

　　它仅有 3 个方法，`initialize`、`get` 和 `close`，但是具有一项比较复杂的功能，即将 BrowserMob Proxy 和 Selenium 捆绑在一起。滚动页面以确保整个页面都被加载，将 HTTP 存档（HAR）文件保存到合适的位置以便后续处理。

consoleservice.py

　　它处理来自控制台的命令，并且负责主 `APIFinder` 类。

harParser.py

　　它解析 HAR 文件并抽取 API 调用。

html_template.html

　　它提供在浏览器中显示 API 调用的一个模板。

README.md

　　Git 的 readme 页面。

从 https://bmp.lightbody.net/ 下载 BrowserMob Proxy 的二进制文件，并将其解压缩文件放到 apiscraper 项目的路径下。

在撰写本书之时 BrowserMob Proxy 的最新版本是 2.1.4，因此我们的代码假定二进制文件放在项目根路径下的 browsermob-proxy-2.1.4/bin/browsermob-proxy 位置。如果有任何变化，你可以在运行时更改路径，或者更简单的办法是修改 apiFinder.py 中对应的代码。

下载 ChromeDriver，并将其放在 apiscraper 项目路径下。

你还需要安装以下 Python 库：

- `tldextract`
- `selenium`
- `browsermob-proxy`

当以上准备工作都完成以后，你可以开始搜集 API 调用了。输入：

```
$ python consoleservice.py -h
```

这会为你提供一系列的选项来开始你的搜集工作：

```
usage: consoleservice.py [-h] [-u [U]] [-d [D]] [-s [S]] [-c [C]] [-i [I]]
                         [--p]

optional arguments:

-h, --help  show this help message and exit
```

```
-u [U]      Target URL. If not provided, target directory will be scanned
            for har files.

-d [D]      Target directory (default is "hars"). If URL is provided,
            directory will store har files. If URL is not provided,
            directory will be scanned.

-s [S]      Search term

-c [C]      File containing JSON formatted cookies to set in driver (with
            target URL only)

-i [I]      Count of pages to crawl (with target URL only)

--p         Flag, remove unnecessary parameters (may dramatically increase
            runtime)
```

你可以搜索针对单个搜索词的单个页面的 API 调用。例如，你可以搜索返回产品数据的 http://target.com 页面的 API：

```
$ python consoleservice.py -u https://www.target.com/p/rogue-one-a-star-wars-\
story-blu-ray-dvd-digital-3-disc/-/A-52030319 -s "Rogue One: A Star Wars Story"
```

上述命令返回的信息包括一个 URL 以及一个返回页面产品数据的 API：

```
URL: https://redsky.target.com/v2/pdp/tcin/52030319
METHOD: GET
AVG RESPONSE SIZE: 34834
SEARCH TERM CONTEXT: c":"786936852318","product_description":{"title":
"Rogue One: A Star Wars Story (Blu-ray + DVD + Digital) 3 Disc",
"long_description":...
```

利用 `-i` 标志位，可以从提供的初始 URL 开始抓取多个页面（默认情况下只有一个页面）。这对于搜索全网流量中特定的关键词，或者通过省略 `-s` 搜索词标志位，抓取每个页面加载时的所有 API 流量非常有用。

所有的抓取数据存储在一个 HAR 文件中，默认放在项目路径下的 /har 文件夹中，而该路径可以通过 `-d` 标志位进行修改。

如果没有提供 URL，你可以传入包含已抓取的 HAR 文件的路径进行查找和分析。

这个项目还提供了很多其他功能，包括：

- 去除非必需的参数（去除 `GET` 或 `POST` 参数，这些参数并不会影响 API 调用的返回值）；
- 多种 API 输出格式（命令行、HTML、JSON）；
- 区分指示单独 API 路由的路径参数和只是作为同一个 API 路由的 `GET` 参数的路径参数。

进一步的发展规划已做好，我和其他人会继续用它进行网页抓取和 API 收集。

12.4　API与其他数据源结合

虽然，许多现代 Web 应用存在的理由就是抓取现有的数据，再用更好看的形式展现出来，但是我觉得这些应用没什么意义。如果你用 API 作为唯一的数据源，那么你最多就是复制别人数据库里的数据，而且这些数据基本上都是已经发表过的。真正有意思的事情，是以一种新颖的方式将两个或多个数据源组合起来，或者把 API 作为一种工具，从全新的视角对抓取到的数据进行解释。

下面介绍如何把 API 和网页抓取结合起来：看看维基百科的贡献者们大都在哪里。

如果你经常用维基百科，可能会注意到词条的编辑历史页面，里面是一列编辑记录。如果用户先登录维基百科再编辑词条，他们的用户名就会显示出来。如果不先登录就对词条进行编辑，他们的 IP 地址就会显示在编辑历史中，如图 12-2 所示。

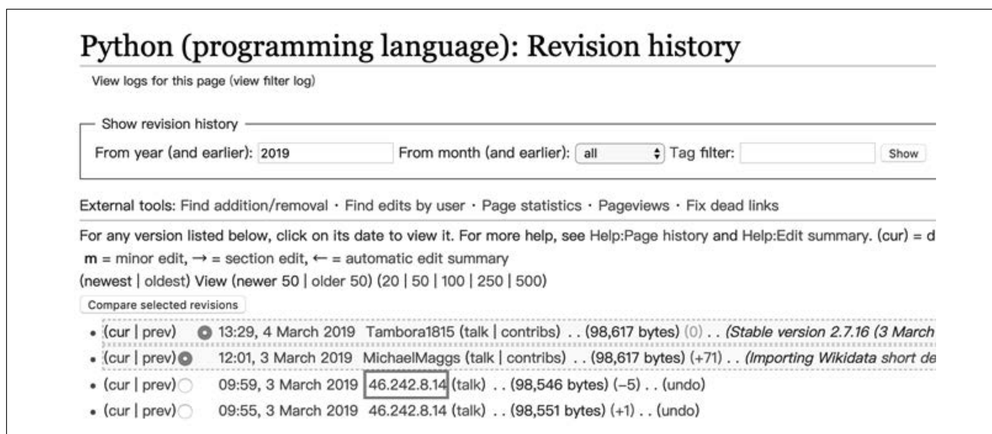


图 12-2：维基百科 Python 词条的编辑历史页面的匿名编辑者的 IP 地址

上图中标注的 IP 地址是 46.242.8.14。写作本书时，利用淘宝 IP 地址库的 API，可以查出这个 IP 地址的地理位置（IP 地址有时会改变地理位置）是俄罗斯的莫斯科市。

一个这样的 IP 地址并没什么意义，但是如果我们可以收集大量维基百科编辑者的地理数据呢？几年前我做过这件事，当时用 Google 的地理图形库（Geochart ）做了一个显示维基百科英文版的编辑者所在位置的可视图，后来又做了其他语言的版本。

首先创建一个抓取维基百科的基本程序，寻找编辑历史页面，然后把其中的 IP 地址找出来，这并不难。只要对第 3 章的代码做些修改就可以，代码如下所示。

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import json
import datetime
```

```
import random
import re

random.seed(datetime.datetime.now())
def getLinks(articleUrl):
    html = urlopen('http://en.wikipedia.org{}'.format(articleUrl))
    bs = BeautifulSoup(html, 'html.parser')
    return bs.find('div', {'id':'bodyContent'}).findAll('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))

def getHistoryIPs(pageUrl):
    # 编辑历史页面的URL链接格式是：
    # http://en.wikipedia.org/w/index.php?title=Title_in_URL&action=history
    pageUrl = pageUrl.replace('/wiki/', '')
    historyUrl = 'http://en.wikipedia.org/w/index.php?title={}&action=history'
        .format(pageUrl)
    print('history url is: {}'.format(historyUrl))
    html = urlopen(historyUrl)
    bs = BeautifulSoup(html, 'html.parser')
    # 找出class属性是"mw-userlink mw-anonuserlink"的链接
    # 它们用IP地址代替用户名
    ipAddresses = bs.findAll('a', {'class':'mw-anonuserlink'})
    addressList = set()
    for ipAddress in ipAddresses:
        addressList.add(ipAddress.get_text())
    return addressList

links = getLinks('/wiki/Python_(programming_language)')

while(len(links) > 0):
    for link in links:
        print('-'*20)
        historyIPs = getHistoryIPs(link.attrs['href'])
        for historyIP in historyIPs:
            print(historyIP)

    newLink = links[random.randint(0, len(links)-1)].attrs['href']
    links = getLinks(newLink)
```

这个程序包含两个函数：getLinks（第 3 章里用过）和新的函数 getHistoryIPs，后者搜索所有 class 属性为 mw-anonuserlink 的链接内容（匿名用户的 IP 地址，而不是用户名），返回一个链接列表。

上面的代码还用了一种随机的（不过对这个示例是有效的）搜索模式来查找词条的编辑历史。首先获取起始词条（示例中是 Python programming language 词条）链接到的所有词条的编辑历史。然后，随机选择一个词条作为起始点，再获取这个页面链接到的所有词条的编辑历史。重复这个过程，直到某个页面不包含其他维基词条的链接为止。

现在，我们获得了编辑历史的 IP 地址数据，把它们与上一节的 getCountry 函数结合起来，就可以查询 IP 地址所属的国家了。我对 getCountry 函数做了一点儿修改，处理了会引起

“404 Not Found”异常的无效或错误的 IP 地址（比如，写作本书时，淘宝 IP 地址库不能查询 IPv6 地址，这可能会引起 404 错误）：

```
def getCountry(ipAddress):
    try:
        response = urlopen('http://ip.taobao.com/service/getIpInfo.php?ip={}'
            .format(ipAddress)).read().decode('utf-8')
        responseJson = json.loads(response)
        country = responseJson.get('data')['country']
    except:
        returnNone
    else:
        return country

links = getLinks('/wiki/Python_(programming_language)')

while(len(links) > 0):
    for link in links:
        print('-'*20)
        historyIPs = getHistoryIPs(link.attrs["href"])
        for historyIP in historyIPs:
            country = getCountry(historyIP)
            if country is not None:
                print('{} is from {}'.format(historyIP, country))

    newLink = links[random.randint(0, len(links)-1)].attrs['href']
    links = getLinks(newLink)
```

下面是部分输出结果：

```
--------------------
history urlis: http://en.wikipedia.org/w/index.php?title=Programming_
paradigm&action=history
117.221.183.123isfrom印度
68.151.180.83isfrom加拿大
129.7.106.20isfrom美国
49.197.5.59isfrom澳大利亚
31.223.170.65isfrom荷兰
174.254.128.149isfrom美国
192.159.69.162isfrom美国
192.117.105.47isfrom以色列
213.133.47.254isfrom荷兰
```

12.5 再说一点API

本章介绍了几种常见的利用现代 API 获取网络数据的方式，以及如何用这些 API 构建快速且强大的网络爬虫。如果你要构建 API 而不仅仅是使用 API，或者如果你希望更多地了解 API 的构建和语法，我推荐你阅读 Leonard Richardson、Mike Amundsen 和 Sam Ruby 合著的 *RESTful Web APIs*。该书针对 Web API 的用法提供了非常全面的理论介绍与实践指导。另外，Mike Amundsen 的精彩视频教学课程“Designing APIs for the Web”，也可以教你创

建自己的 API。如果你想用一种便捷的格式分享自己抓取的数据，他的视频非常有用。

尽管有些人可能抱怨当前 JavaScript 和动态网站盛行，使得传统的“抓取并解析 HTML 页面”的方法过时了，我个人却非常欢迎这种新趋势。动态网站更多地依赖有 JSON 格式的 HTML 文件，而较少依赖人工编写的 HTML，这为所有希望获得简洁且格式友好的数据的人提供了福利。

Web 不再是偶尔带有一些多媒体和 CSS 样式的 HTML 页面集合，而是上百种文件类型和数据格式的集合，通过浏览器一次进行上百次高速数据传输来加载你需要的所有页面内容。真正的技巧通常是透彻地认识你眼前的网页，并直接从数据源获取数据。

第13章

图像识别与文字处理

从 Google 的无人驾驶汽车到可以识别假钞的自动售卖机，机器视觉是应用广泛且具有深远影响的一大领域。这一章将重点介绍机器视觉的一个分支——文字识别，介绍如何用一些 Python 库来识别和使用基于文字的图像。

当你不想让自己的文字被网络机器人抓取时，把文字做成图片放在网页上是常用的办法。在联系人表单里经常可以看到一个邮箱地址被部分或全部转换成图片。人们可能觉察不出明显的差异，但是机器人阅读这些图片会非常困难，这种方法可以防止多数垃圾邮件发送器轻易地获取你的邮箱地址。

当然，验证码（CAPTCHA）就利用了这种人类用户可以正常读取但是大多数机器人都无法读取的图片。验证码的读取难度不同，有些验证码比其他的更加难读，后面会介绍这个问题。

但是，验证码并不是网络爬虫抓取数据时需要进行图像转文字工作的唯一对象。即便是今天，仍然有很多文档是扫描后直接放到网上的，它们无法直接使用，尽管"近在眼前"。如果无法将图像转为文字，要想使用这些文档的内容，就只能人工手敲了，可没人愿意花时间干这事儿。

将图像转化成文字被称为**光学字符识别**（optical character recognition，OCR）。可以实现 OCR 的底层库并不多，目前很多库都是使用几个共同的底层 OCR 库，或者是在上面进行定制。这类 OCR 系统有时会变得非常复杂，所以我建议你先阅读本章第一节的内容，再实践这一章的代码示例。

13.1 OCR库概述

对于图像读取和处理、图像相关的机器学习以及图像创建等任务来说，Python 是一门非常出色的语言。虽然有很多库可以进行图像处理，但这里只重点介绍两个库：Pillow 和 Tesseract。

这两个库互为补充，共同对互联网上的图片进行处理和 OCR 识别。Pillow 执行第一步，清洗和过滤图像，而 Tesseract 尝试将图像中的形状与库里面存储的文字相匹配。

本章将介绍这两个库的安装方法和基本用法，并给出这两个库配合使用的几个示例。此外还将介绍一些高级的 Tesseract 训练，以便你训练 Tesseract 识别你在网上遇到的其他字体和语言（甚至是 CAPTCHA）。

13.1.1 Pillow

尽管 Pillow 算不上是图像处理功能最全的库，但是它拥有你需要使用的全部功能，除非你要用 Python 重写一个 Photoshop。它也是一个文档健全且十分易用的库。

Pillow 是从 Python 2.x 的 Python 图像库（Python Imaging Library，PIL）分出来的，支持 Python 3.x。和 PIL 一样，Pillow 也可以轻松地导入图片，并通过大量的过滤、修饰甚至像素级的变换处理图片：

```
from PIL import Image, ImageFilter

kitten = Image.open('kitten.jpg')
blurryKitten = kitten.filter(ImageFilter.GaussianBlur)
blurryKitten.save('kitten_blurred.jpg')
blurryKitten.show()
```

在上面这个例子中，图片 kitten.jpg 会在默认的图片浏览器里打开，不过看着会有点儿模糊。之后，这个比较模糊的图片被另存为 kitten_blurred.jpg，与原图放在一个文件夹里。

我们可以用 Pillow 完成图片的预处理，让机器更方便地读取图片。但是如前所述，除了这些简单的过滤工作之外，Pillow 还可以完成许多复杂的图像处理工作。更多的信息，请查看 Pillow 文档。

13.1.2 Tesseract

Tesseract 是一个 OCR 库，目前由 Google（一家以 OCR 和机器学习技术闻名于世的公司）赞助。Tesseract 是目前公认最优秀、最精确的开源 OCR 系统。

除了极高的精确度，Tesseract 还具有很高的灵活性。它可以通过训练识别出任何字体（只要这些字体的风格保持不变就可以，后面会介绍），也可以识别出任何 Unicode 字符。

本章既会使用命令行程序 Tesseract，也会用到第三方 Python 包装器 pytesseract。两者的命名非常清楚，因此当你看到“Tesseract”时，我指的是命令行软件，而当你看到“pytesseract”时，我指的是第三方 Python 包装器。

1. 安装Tesseract

在 Windows 系统上，下载方便的可执行安装文件安装即可。写作本书时，Tesseract 的最新版本是 3.02，更新的版本应该也可以这样安装。

Linux 用户可以通过 `apt-get` 安装：

```
$ sudo apt-get install tesseract-ocr
```

在 Mac 上安装 Tesseract 有点儿复杂，不过用 Homebrew 等第三方库可以很方便地安装。Homebrew 在第 6 章介绍 MySQL 安装过程时提到过。例如，你可以用下面两行代码首先安装 Homebrew，然后再安装 Tesseract：

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/ \
            install/master/install)"
$ brew install tesseract
```

也可以从 Tesseract 项目的下载页面下载源代码安装。

要使用 Tesseract 的某些功能，比如在后面的示例中训练程序识别新字符，你需要先在系统中设置一个新的环境变量 `$TESSDATA_PREFIX`，让 Tesseract 知道训练的数据文件存储在哪里。

在大多数 Linux 系统和 macOS 系统上，你可以这么设置：

```
$ export TESSDATA_PREFIX=/usr/local/share/
```

值得注意的是，虽然 /usr/local/share/ 是 Tesseract 的默认数据存储位置，但你还是应该仔细地检查一下，确保自己的安装没问题。

类似地，在 Windows 系统上，你可以通过下面这行命令设置环境变量：

```
# setx TESSDATA_PREFIX C:\Program Files\Tesseract OCR\
```

2. pytesseract

安装好 Tesseract 后，你就可以着手安装 Python 包装器库 `pytesseract` 了，它会利用已安装好的 Tesseract 读取图像文件并输出字符串和可用在 Python 代码中的对象。



代码示例需要 pytesseract 0.1.9

需要注意的是，在 pytesseract 版本 0.1.8 和 0.1.9 之间，作者做了很大的改变。本节将仅仅用到该库 0.1.9 版本的功能。请确保在运行本章的示例代码前安装好正确的版本。

你可以通过 pip 安装 pytesseract，或者从 pytesseract 项目的页面下载并运行：

```
$ python setup.py install
```

可以结合 PIL 使用 pytesseract，以从图像中读取文字：

```
from PIL import Image
import pytesseract

print(pytesseract.image_to_string(Image.open('files/test.png')))
```

如果你的 Tesseract 库安装在你的 Python 路径下，你可以对 pytesseract 进行如下设置：

```
pytesseract.pytesseract.tesseract_cmd = '/path/to/tesseract'
```

除了返回图像的 OCR 结果外，pytesseract 还有一些有用的功能。它可以估计边界（每个字符的边界的像素位置）：

```
print(pytesseract.image_to_boxes(Image.open('files/test.png')))
```

它还会返回所有数据的完整输出，例如置信分数、页数、行数、像素位置数据以及其他信息：

```
print(pytesseract.image_to_data(Image.open('files/test.png')))
```

默认情况下，最后两个输出文件是用空格或 tab 分隔的字符串文件，但是你也可以输出字典，或者在不能够进行 UTF-8 解码的情况下输出字节字符串：

```
from PIL import Image
import pytesseract
from pytesseract import Output

print(pytesseract.image_to_data(Image.open('files/test.png'),
    output_type=Output.DICT))
print(pytesseract.image_to_string(Image.open('files/test.png'),
    output_type=Output.BYTES))
```

本章会将 pytesseract 库和命令行 Tesseract 结合起来使用，并从 Python 通过 `subprocess` 库触发 Tesseract。尽管 pytesseract 库很实用也很方便，但是它仍然实现不了 Tesseract 的部分功能，因此最好是熟悉所有这些方法。

13.1.3 NumPy

虽然 NumPy 并非解决 OCR 问题时必须使用的库，但是如果你想训练 Tesseract 识别本章后面提到的字符或字体，那么就会用到它。在后面的一些代码示例中，你也会用它来完成简单的数学任务（如计算加权平均值）。

NumPy 是一个非常强大的库，具有大量线性代数以及大规模科学计算的方法。因为 NumPy 可以用数学方法把图片表示成巨大的像素数组，所以它可以流畅地配合 Tesseract 完成任务。

和其他 Python 库一样，NumPy 可以通过第三方包管理器（比如 pip）或者通过下载包利用 `$python setup.py install` 来安装。

即使你不打算运行这里使用 NumPy 的任何示例代码，我也强烈建议你安装 NumPy 或者将其加入到你的 Python“武器库”中。它可以替代 Python 内置的数学库并且有很多实用的功能，特别是数组的运算操作。

通常情况下，NumPy 的导入和使用方式如下所示：

```
import numpy as np

numbers = [100, 102, 98, 97, 103]
print(np.std(numbers))
print(np.mean(numbers))
```

这段代码打印出了一组数的标准差和平均值。

13.2　处理格式规范的文字

运气好的话，你要处理的大多数文字都是比较干净、格式规范的。格式规范的文字通常可以满足一些需求，不过究竟什么算“格式混乱”，什么算“格式规范”，确实因人而异。

通常，格式规范的文字具有以下特点：

- 使用一种标准字体（不包含手写体、草书，或者十分“花哨的”字体）
- 虽然是复印的或是照片，字体还是很清晰，没有多余的痕迹或污点
- 排列整齐，没有歪歪斜斜的字
- 没有超出图片范围，也没有残缺不全或紧紧贴在图片的边缘

文字的一些格式问题可在图片预处理时解决。例如，可以把图片转换成灰度图，调整亮度和对比度，还可以根据需要进行裁剪和旋转。但是，有些基本的限制可能要求进行更广泛的训练。详情请见 13.3 节。

图 13-1 是格式规范文字的一个理想示例。

This is some text, written in Arial, that will be read by
Tesseract. Here are some symbols: !@#$%^&*()

图 13-1：样本文字被保存为 .tiff 文件，将由 Tesseract 读取

你可以通过下面的命令运行 Tesseract，读取文件并把结果写到一个文本文件中：

```
$ tesseract text.tif textoutput | cat textoutput.txt
```

输出结果的第一行是 Tesseract 的版本信息，表明它正在运行，后面是图片识别结果 textoutput.txt 文件里的内容：

```
Tesseract Open Source OCR Engine v3.02.02 with Leptonica
This is some text, written in Arial, that will be read by
Tesseract. Here are some symbols: !@#$%"&'()
```

你会发现识别结果很准确，不过符号“^”和“*”分别被解释成了双引号和单引号。大体上，你可以很舒服地阅读。

对图片进行模糊处理，转换成一个 JPG 压缩格式的图片，再增加一点儿背景渐变，识别效果就会变得很差（如图 13-2 所示）。

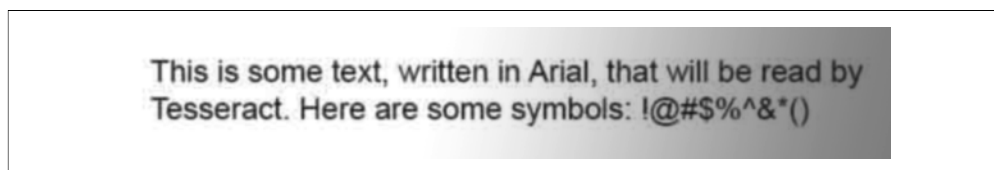


图 13-2：你在网上看到的许多图片可能都像这样

Tesseract 不能完整处理这个图片，主要是因为图片背景色是渐变的，最终结果是这样：

```
This is some text, written In Arlal, that"
Tesseract. Here are some symbols: _
```

你会发现，随着背景色从左到右不断加深，文字变得越来越难以识别，Tesseract 识别出的每一行的最后一个字符都是错的。另外，经过 JPG 格式转换和模糊效果处理，Tesseract 更难区分小写“i”和大写“I”以及数字“1”。

遇到这类问题，可以先用 Python 脚本对图片进行清理。利用 Pillow 库，你可以创建一个阈值过滤器来去掉渐变的背景色，只把文字留下来，从而让图片更加清晰，便于 Tesseract 读取。

另外，除了从命令行使用 Tesseract，你也可以使用 pytesseract 库来运行 Tesseract 命令并读取结果文件。

```
from PIL import Image
import pytesseract

def cleanFile(filePath, newFilePath):
    image = Image.open(filePath)
```

```
    # 为图像设置一个阈值过滤器并保存
    image = image.point(lambda x: 0 if x < 143 else 255)
    image.save(newFilePath)
    return image

image = cleanFile('files/textBad.png', 'files/textCleaned.png')

# 调用Tesseract对新创建的图像进行OCR识别
print(pytesseract.image_to_string(image))
```

程序自动创建的 textCleaned.png 如图 13-3 所示。

This is some text, written in Arial, that will be read by
Tesseract Here are some symbols: !@#$%^&*()

图 13-3：通过一个阈值对前面的"模糊"图片进行过滤的结果

除了一些标点符号不太清晰或丢失了，大部分文字都是可读的，至少对我们而言如此。Tesseract 给出了其最好的结果：

```
This us some text' written In Anal, that will be read by
Tesseract Here are some symbols: !@#$%"&'()
```

图片上的句点和逗号经过处理后变得非常小，不论从我们的视角还是 Tesseract 的视角看，它们都从图片上基本消失了。还有一点失误是把"Arial"解释成了"Anal"，这是 Tesseract 把"r"和"i"都解释成了"n"的结果。

不过，相比上一版本被截断的识别结果，这一版算是有很大进步了。

Tesseract 最大的弱项是对渐变背景色的处理。在之前那个版本中，Tesseract 的算法在读取文字之前会自动尝试调整图片对比度，但是如果你用 Pillow 库这样的工具对图片进行预处理，效果会更好。

在提交给 Tesseract 处理之前，那些带标题的、带有大片空白的图片，或者有其他问题的图片，都应该进行预处理。

13.2.1　自动调整图像

在前面的例子中，值 143 作为理想的阈值来将图像像素调整成黑色或者白色，这样 Tesseract 就可以读取图像了。但是如果你有很多图像，它们全都有不同程度的灰度问题，无法手动一一调整怎么办？

寻找最佳（至少是非常好的）解决方案的一种方式，是对一些调整到不同值的图像运行 Tesseract，并利用算法选择结果最好的那个值，结果可以通过 Tesseract 能够读取的字符 /

字符串数量以及 Tesseract 读取这些字符时使用的“置信值”的某种组合来度量。

具体使用哪种算法会因应用的不同而略有差异，但以下是一个对不同图像处理阈值进行迭代，以便找到“最佳”设置的示例。

```
import pytesseract
from pytesseract import Output
from PIL import Image
import numpy as np

def cleanFile(filePath, threshold):
    image = Image.open(filePath)
    # 为图像设置一个阈值过滤器并保存
    image = image.point(lambda x: 0 if x < threshold else 255)
    return image

def getConfidence(image):
    data = pytesseract.image_to_data(image, output_type=Output.DICT)
    text = data['text']
    confidences = []
    numChars = []

    for i in range(len(text)):
        if data['conf'][i] > -1:
            confidences.append(data['conf'][i])
            numChars.append(len(text[i]))

    return np.average(confidences, weights=numChars), sum(numChars)

filePath = 'files/textBad.png'

start = 80
step = 5
end = 200

for threshold in range(start, end, step):
    image = cleanFile(filePath, threshold)
    scores = getConfidence(image)
    print("threshold: " + str(threshold) + ", confidence: "
        + str(scores[0]) + " numChars " + str(scores[1]))
```

上述代码中有两个函数。

`cleanFile`

输入原始的“坏”文件并用一个阈值变量运行 PIL 阈值工具。它处理文件并返回 PIL 图像对象。

`getConfidence`

输入清洗后的 PIL 图像对象并通过 Tesseract 运行。它计算每个识别字符串的平均置信值（通过字符串中字符的数量统计），以及识别字符的数量。

通过改变阈值，获得识别字符的置信值和数量，得到如下输出：

```
threshold: 80, confidence: 61.8333333333 numChars 18
threshold: 85, confidence: 64.9130434783 numChars 23
threshold: 90, confidence: 62.2564102564 numChars 39
threshold: 95, confidence: 64.5135135135 numChars 37
threshold: 100, confidence: 60.7878787879 numChars 66
threshold: 105, confidence: 61.9078947368 numChars 76
threshold: 110, confidence: 64.6329113924 numChars 79
threshold: 115, confidence: 69.7397260274 numChars 73
threshold: 120, confidence: 72.9078947368 numChars 76
threshold: 125, confidence: 73.582278481 numChars 79
threshold: 130, confidence: 75.6708860759 numChars 79
threshold: 135, confidence: 76.8292682927 numChars 82
threshold: 140, confidence: 72.1686746988 numChars 83
threshold: 145, confidence: 75.5662650602 numChars 83
threshold: 150, confidence: 77.5443037975 numChars 79
threshold: 155, confidence: 79.1066666667 numChars 75
threshold: 160, confidence: 78.4666666667 numChars 75
threshold: 165, confidence: 80.1428571429 numChars 70
threshold: 170, confidence: 78.4285714286 numChars 70
threshold: 175, confidence: 76.3731343284 numChars 67
threshold: 180, confidence: 76.7575757576 numChars 66
threshold: 185, confidence: 79.4920634921 numChars 63
threshold: 190, confidence: 76.0793650794 numChars 63
threshold: 195, confidence: 70.6153846154 numChars 65
```

从结果中可以看出平均置信值以及识别出的字符数量的变化规律。这两个值都在阈值 145 附近达到最高点，这个阈值与我们手动发现的 143 这个“理想”阈值非常接近。

140 和 145 这两个阈值给出了最大识别字符数量（83），但是阈值 145 给出了这些字符的最大置信值，因此你可能希望采用这个结果，并返回该阈值对应的识别文本作为对图像所包含文字的“最佳猜测”。

当然，仅仅找到“最多”的字符并不意味着所有这些字符都是真实存在的。取某些阈值时，Tesseract 可能会将单个字符分成多个字符，或者将图像中的随机噪声解释成一个实际上不存在的文字字符。在这种情况下，你可能需要更多地参考每个分数的平均置信值。

例如，如果你看到的部分结果如下：

```
threshold: 145, confidence: 75.5662650602 numChars 83
threshold: 150, confidence: 97.1234567890 numChars 82
```

你很容易采用得出这样的结论：阈值 150 提高了 20% 的置信率，而仅仅损失了一个字符，那么阈值 145 肯定是不准确的，或者是分割了某个字符，或者是发现了一个不存在的字符。

这时，提前进行实验，以完善你的阈值选择算法就很有用了。例如，你可能要选择使置信率和字符数量的**乘积**最大的分数（在上面的例子中，仍然是阈值 145 获胜，对应的乘积是 6272，而对于我们假想的例子来说，是阈值 150 获胜，其对应的乘积是 7964）或者其他指标。

注意，这种选择算法对于阈值以外的其他 PIL 工具值依然是有效的。你也可以用它选择两个或者两个以上不同的值，并用同样的方式选择最佳的结果分数。

显然，这种选择算法是计算密集型的。你需要对每一幅图像多次运行 PIL 和 Tesseract，而如果你提前知道“理想”阈值的话，就仅仅需要运行一次。

还需要记住的是，当开始处理图像时，你可能会开始注意到“理想”阈值的模式。你可能只需要尝试 130 至 180 区间的阈值，而不需要尝试在 80 至 200 之间的每一个阈值。

你甚至可能尝试另外一种方法，即在第一轮选择其中 20 个阈值，然后用贪婪算法寻找最佳结果。具体做法是在前一轮迭代中发现的“最佳”阈值之间逐步减小步长。当你处理多个变量时，这种方法也是非常适用的。

13.2.2　从网站图片中抓取文字

用 Tesseract 读取硬盘里图片上的文字可能不怎么令人兴奋，但当我们将它和网络爬虫结合起来使用时，就会变成一个强大的工具。网站上的图片可能并不是故意把文字弄得模糊难认（就像当地餐厅网站上菜单的 JPG 图片一样），但它们也可以故意隐藏文字，如下一个例子所示。

虽然亚马逊的 robots.txt 文件允许抓取网站的产品页面，但是图书的预览页通常无法抓取。这是因为图书的预览页是通过用户触发的 Ajax 脚本进行加载的，预览图片隐藏在 div 节点下面。对于普通的网站访问者来说，它们看起来更像是 Flash 动画，而不是图像文件。当然，即使我们能获得图片，要把它们读成文字也没那么简单。

下面的程序就解决了这个问题：首先导航到托尔斯泰的《战争与和平》的大字号印刷版[1]，打开阅读器，收集图片的 URL 链接，然后下载图片，识别图片，最后打印每个图片中的文字。

请注意，这段代码的正确运行取决于真实的亚马逊列表以及亚马逊网站的一些架构特征。如果这个列表被替换下去了，你可以用另外一本书的预览 URL 替换（我发现放大显示的时候，sans-serif 字体也能正常显示）。

因为这个程序很复杂，利用了前面几章的多个程序片段，所以我增加了一些注释，以让每段代码的目的更加清晰。

```
import time
from urllib.request import urlretrieve
```

注 1：当处理那些没有训练过的文字时，Tesseract 对大字号印刷版图书的识别效果更好，尤其是图片比较小的时候。下一节将介绍如何用不同的字体训练 Tesseract，这样可以帮助它识别更小的字，包括普通字号印刷版图书。

```
from PIL import Image
import tesseract
from selenium import webdriver


def getImageText(imageUrl):
    urlretrieve(image, 'page.jpg')
    p = subprocess.Popen(['tesseract', 'page.jpg', 'page'],
        stdout=subprocess.PIPE,stderr=subprocess.PIPE)
    p.wait()
    f = open('page.txt', 'r')
    print(f.read())

# 创建新的Selenium driver
driver = webdriver.Chrome(executable_path='<Path to chromedriver>')

driver.get('https://www.amazon.com/Death-Ivan-Ilyich'\
    '-Nikolayevich-Tolstoy/dp/1427027277')
time.sleep(2)

# 点击图书预览按钮
driver.find_element_by_id('imgBlkFront').click()
imageList = []

# 等待页面加载
time.sleep(5)

while 'pointer' in driver.find_element_by_id(
    'sitbReaderRightPageTurner').get_attribute('style'):
    # 当右箭头可以点击时，点击翻页
    driver.find_element_by_id('sitbReaderRightPageTurner').click()
    time.sleep(2)
    # 获取已加载的任何新页面（可以同时加载多个页面，
    # 但是由于使用的是集合，重复的页面不会被加进来）
    pages = driver.find_elements_by_xpath('//div[@class=\'pageImage\']/div/img')
    if not len(pages):
        print("No pages found")
    for page in pages:
        image = page.get_attribute('src')
        print('Found image: {}'.format(image))
        if image not in imageList:
            imageList.append(image)
            getImageText(image)

driver.quit()
```

尽管在理论上上述代码可以用任意类型的 Selenium WebDriver 运行，但我发现目前在 Chrome 上运行最稳定。

正如你之前在 Tesseract 阅读器中体验过的一样，上述代码将打印很长一段书的内容，即第一章的内容：

```
Chapter I

During an Interval In the Melvmskl trial In the large
building of the Law Courts the members and public
prosecutor met in [van Egorowch Shebek's private
room, where the conversation turned on the celebrated
Krasovski case. Fedor Vasillevich warmly maintained
that it was not subject to their jurisdiction, Ivan
Egorovich maintained the contrary, while Peter
ivanowch, not havmg entered into the discussmn at
the start, took no part in it but looked through the
Gazette which had Just been handed in.

"Gentlemen," he said, "Ivan Ilych has died!"
```

当然，其中很多单词都存在明显的错误，例如“Melvmsl”应该是“Melvinski”，“discussmn”应该是“discussion”。很多这种错误可以根据词典单词列表进行猜测（或许有些基于专有名词，如“Melvinski”）。

偶尔，一个错误可能囊括了整个单词，例如第 3 页的文本：

```
it is he who is dead and not 1.
```

在这个例子中，单词“I”被识别成字符“1”。这里可以使用马尔可夫链分析以及单词词典来解决这个问题。如果文本的任何部分包含非常不常见的短语（如“and not 1”），就可以认为该文本实际上应该是更常见的“and not I”。

当然，这些字符的替换也应该遵循一定的可预测模式：“vi”变成“w”，“I”变成“1”。如果这些替换频繁地出现在你的文本中，你可以创建一个列表来“尝试”出新的单词和短语，选择更合理的解决方案。一种方法是替换掉经常被混淆的字符，并用字典中的单词去匹配，或者用已识别出的（或最常见的）n-gram 匹配。

如果你采用这种方法，请阅读第 9 章了解更多关于文本和自然语言处理的信息。

尽管在这个例子中文本是常见的 sans-serif 字体，Tesseract 应该可以轻易识别出来，但有时候重新训练可以进一步提高准确率。下一节将介绍另一种方法来解决文字混乱的问题。

通过给 Tesseract 提供已知的文字与图片映射集，经过训练 Tesseract 就可以“学会”识别同一种字体，而且可以达到更高的精确率和准确率，哪怕图片中的文字有背景色和相对位置等问题。

13.3　读取验证码与训练Tesseract

虽然大多数人对单词“CAPTCHA”都很熟悉，但是很少有人知道它的具体含义：全自动区分计算机和人类的图灵测试（Completely Automated Public Turing Test to Tell Computers

and Humans Apart）。它的奇怪缩写似乎暗示它一直在扮演着十分奇怪的角色。其目的是为了阻止网站访问，而不是让访问更通畅，它经常让人类和非人类的网络机器人深陷验证码识别的泥潭。

艾伦·图灵在 1950 年发表的论文“Computing Machinery and Intelligence”中首次描述了图灵测试。他在论文中描述了这样一种场景：一个人可以通过计算机终端与其他人以及人工智能程序交流。如果一番对话之后，这个人无法区分出人和人工智能程序，那么就认为这个人工智能程序通过了图灵测试，图灵认为这个人工智能程序就可以真正地“思考”所有的事情。

令人啼笑皆非的是，在过去的 60 多年里，我们从使用这些测试来测试机器，变成了用它们来测试我们自己，结果喜忧参半。Google 近来放弃了其难得令人发指的 reCAPTCHA，主要是因为它也拦截了正常的人类用户。[2]

大多数其他的验证码都是比较简单的。例如，流行的 PHP 内容管理系统 Drupal 有一个著名的验证码模块，可以生成不同难度的验证码。默认图片如图 13-4 所示。

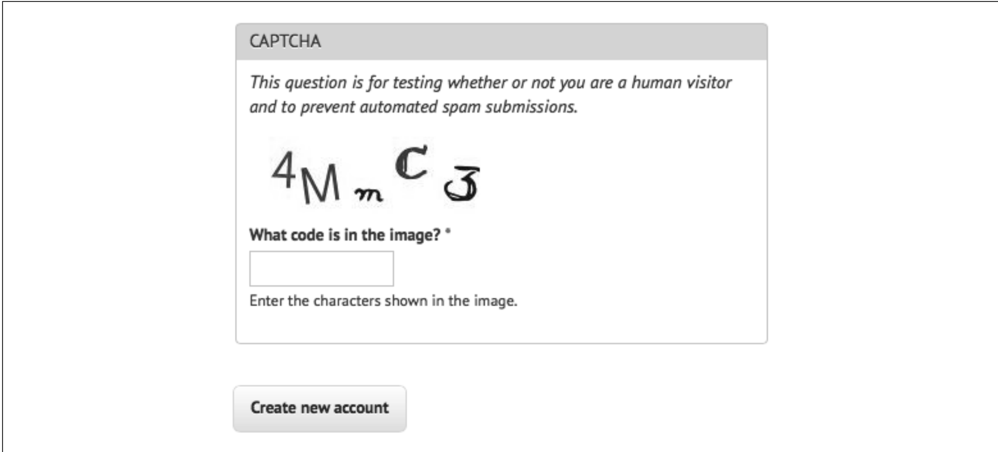


图 13-4：Drupal 验证码项目的默认文字验证码示例

那么与其他验证码相比，究竟是什么让这个验证码更容易被人类和机器读懂呢？

- 字符没有叠加在一起，在水平方向上也没有交叉。也就是说，可以在每一个字符外面画一个方框，而不会与其他字符重叠。
- 没有背景图、线条或其他会对 OCR 程序产生干扰的噪点。
- 虽然在图中不明显，但是这个验证码用的字体种类很少，而且用的是 sans-serif 字体（如“4”和“M”）和一种手写体（如“m”“C”和“3”）。

注 2：详情请见 https://gizmodo.com/google-has-finally-killed-the-captcha-1793190374。

- 白色背景与深色字母之间的对比度很高。

这个验证码只做了一点点改变，就让 OCR 程序很难识别。

- 字母和数字都使用了，这会增加待搜索字符的数量。
- 字母随机的倾斜程度会迷惑 OCR 软件，但是人类还是很容易识别的。
- 那个比较陌生的手写字体很有挑战性，其中"C"和"3"里面有额外的线条。另外，对于非常小的小写"m"，计算机需要进行额外的训练才能识别。

用下面的代码运行 Tesseract 识别图片：

```
$ tesseract captchaExample.png output
```

得到的 output.txt 文件是：

```
4N\,,,C<3
```

虽然识别出了 4、C 和 3，但是显然无法很快识别出正确的验证码。

训练Tesseract

要训练 Tesseract 识别一种文字，无论是晦涩难懂的字体还是验证码，你都需要向 Tesseract 提供每个字符不同形式的样本。

这项枯燥的工作可能要花好几个小时，你可能更想用这个时间找个好看的视频或电影看看。首先要把验证码的多个样本下载到一个文件夹里。下载的样本数量取决于验证码的复杂程度；我在训练集里一共放了 100 个样本（一共 500 个字符，平均每个字符 8 个样本；a-z 的大小写字母加数字 0-9，一共 62 个字符），应该足够训练的了。



建议使用验证码的真实结果给每个样本文件命名（如 4MmC3.jpg）。这有助于你一次性对大量的文件进行快速检查——你可以先把图片调成缩略图模式，然后通过文件名对比不同的图片。这样，在后面的步骤中进行训练效果的检查也会很方便。

第二步是准确地告诉 Tesseract 一张图片中的每个字符是什么，以及每个字符的具体位置。这里需要创建一些矩形定位文件（box file），为每个验证码图片生成一个矩形定位文件。一个图片的矩形定位文件如下所示：

```
4 15 26 33 55 0
M 38 13 67 45 0
m 79 15 101 26 0
C 111 33 136 60 0
3 147 17 176 45 0
```

第一列符号是图片中的每个字符，后面的 4 个数字分别是包围这个字符的最小矩形的坐标[3]，最后一个数字“0”表示图片样本的编号。

显然，手工创建这些图片的矩形定位文件很无聊，不过有一些工具可以帮上忙。我很喜欢在线工具 Tesseract OCR Chopper，因为它不需要安装，也没有其他依赖，只要有浏览器就可以运行，而且用法很简单：上传图片，如果要增加新矩形就单击“Add”按钮，还可以根据需要调整矩形的尺寸，最后把新生成的矩形定位文件复制到一个新文件里就可以了。

矩形定位文件必须保存在一个以 .box 为后缀的纯文本文件中。和图片文件一样，文本文件也用验证码的实际结果命名（例如，4MmC3.box）。同样，这样便于检查 .box 文件的内容和文件的名称，而且按文件名对目录中的文件排序之后，就可以将 .box 文件与对应的图片文件的实际结果进行对比。

你需要创建大约 100 个 .box 文件来保证你有足够的训练数据。因为 Tesseract 有时会忽略那些不能读取的文件，所以建议你尽量多做一些矩形定位文件，以保证训练数据足够充分。如果你觉得训练的 OCR 结果没有达到你的期望，或者 Tesseract 识别某些字符时总是出错，多创建一些训练数据然后重新训练将是一个不错的改进方法。

创建完满载 .box 文件和图片文件的数据文件夹之后，在做进一步分析之前最好备份一下这个文件夹。虽然在数据上运行训练程序不太可能删除任何数据，但是创建 .box 文件花了你好几个小时的时间，来之不易，稳妥一点儿总没错。此外，能够抓取一个满是编译数据的混乱目录，然后再尝试一次，总是好的。

完成所有的数据分析工作和创建 Tesseract 所需的训练文件，一共有 6 个步骤。有一些工具可以帮你处理图片和 .box 文件，不过目前 Tesseract 3.02 还不支持它们。

我写了一个 Python 版的解决方案（https://github.com/REMitchell/tesseract-trainer）来处理同时包含图片文件和 .box 文件的数据文件夹，然后自动创建所有必需的训练文件。

这个解决方案的主要配置方式和步骤都在 `__init__` 方法和 `runAll` 方法里：

```
def __init__(self):
    languageName = 'eng'
    fontName = 'captchaFont'
    directory = '<path to images>'

def runAll(self):
    self.createFontFile()
    self.cleanImages()
    self.renameFiles()
    self.extractUnicode()
```

注 3：图片左下角是原点 (0,0)，4 个数字分别对应每个字符的左下角 *x* 坐标、左下角 *y* 坐标、右上角 *x* 坐标和右上角 *y* 坐标。——译者注

```
            self.runShapeClustering()
            self.runMfTraining()
            self.runCnTraining()
            self.createTessData()
```

你需要动手设置的只有 3 个变量。

`languageName`

Tesseract 用 3 个字母的语言代码表示识别的语言种类。大多数情况下，你可能都会用“eng”表示英语（English）。

`fontName`

表示你选择的字体名称。可以是任意名称，但必须是一个不包含空格的单词。

`directory`

表示包含所有图片和 .box 文件的目录。建议你使用文件夹的绝对路径，如果你使用相对路径，可能需要以 Python 代码运行的目录位置为原点。如果你使用绝对路径，就可以在电脑的任意位置运行代码了。

让我们再看看 `runAll` 里每个函数的用法。

`createFontFile` 创建了一个 font_properties 文件，让 Tesseract 知道你要创建的新字体：

```
captchaFont 0 0 0 0 0
```

这个文件包含字体的名称，后面跟着若干 1 和 0，分别表示应该使用斜体、粗体或其他版本的字体（用这些属性训练字体是一个很好玩儿的练习，不过超出了本书的介绍范围，感兴趣的同学可以自己尝试）。

`cleanImages` 首先创建所有样本图片的高对比度版本，然后转换成灰度图，并进行一些清理，让 OCR 程序更容易读取。如果你要处理的验证码图片上面有一些很容易过滤掉的噪点，那么你可以在这里增加一些步骤来处理它们。

`renameFiles` 把所有的图片文件和 .box 文件的文件名改变成 Tesseract 需要的形式（fileNumber 是文件序号，用来区别每个文件）：

- <languageName>.<fontName>.exp<fileNumber>.box
- <languageName>.<fontName>.exp<fileNumber>.tiff

`extractUnicode` 函数会检查所有已创建的 .box 文件，确定要训练的字符集范围。抽取出的 Unicode 文件会告诉你一共找到了多少个不重复的字符，这也是一个查询字符的好方法，如果你漏了字符可以用这个结果快速排查。

之后的 3 个函数，`runShapeClustering`、`runMfTraining` 和 `runCtTraining`，分别用来创建文

件 shapetable、pfftable 和 normproto。它们会生成每个字符的几何和形状信息，也会提供统计信息，以便 Tesseract 计算给定字符是某种类型的概率。

最后，Tesseract 会用之前设置的语言名称，对数据文件夹编译出的每个文件进行重命名（例如，shapetable 被重命名为 eng.shapetable），然后把所有的文件编译到最终的训练数据文件 eng.traineddata 中。

你需要动手完成的唯一步骤，就是用下面的 Linux 和 Mac 命令把刚刚创建的 eng.traineddata 文件复制到 tessdata 文件夹里。

```
$cp /path/to/data/eng.traineddata $TESSDATA_PREFIX/tessdata
```

经过这些步骤之后，你就可以用 Tesseract 训练过的这些验证码来识别新图片了。现在用 Tesseract 重新读取之前的示例验证码图片，就可以得到正确的结果了：

```
$ tesseract captchaExample.png output|cat output.txt
4MmC3
```

成功啦！相比之前的识别结果 4N\,,,C<3，这个识别结果有明显的改善。

前面的内容只是对 Tesseract 库强大的字体训练和识别能力的一个概述。如果你对 Tesseract 的其他训练方法感兴趣，甚至打算建立自己的验证码训练文件库，或者想和全世界的 Tesseract 爱好者分享自己对一种新字体的识别成果，那么我建议你仔细阅读 Tesseract 的文档。

13.4 获取验证码并提交答案

许多流行的内容管理系统即使加了验证码模块，其众所周知的注册页面也经常会遭到网络机器人的垃圾注册。比如在 http://pythonscraping.com/ 上，即使加了验证码（的确也很容易识别）也无法抑制大量的垃圾注册。

那么，这些网络机器人究竟是怎么做的呢？我们已经成功地识别出保存在电脑中的验证码了，那么如何才能实现一个全能的网络机器人呢？本节将综合前面几章的内容来告诉你答案。如果你还没准备好，请至少先浏览一下第 10 章。

大多数网站生成的验证码图片都具有以下属性。

- 它们是服务器端程序动态生成的图片。验证码图片的 src 属性可能和普通图片不太一样，比如 <img src="WebForm.aspx?id=8AP85CQKE9TJ">，但是可以和其他图片一样进行下载和处理。
- 图片的答案存储在服务器端的数据库里。
- 很多验证码都有时效，如果你长时间没识别出来就会失效。虽然这对网络机器人来说不是什么问题，但是如果你想保留验证码的答案一会儿再使用，或者想通过一些方法延长验证码的有效时限，很难成功。

常用的处理方法是，首先把验证码图片下载到硬盘里，清理干净，然后用 Tesseract 处理图片，最后返回符合网站要求的识别结果。

我在 http://pythonscraping.com/humans-only 创建了一个带验证码的评论表单，来演示如何用网络机器人破解验证码。该网络机器人使用命令行 Tesseract 库，而不是 pytesseract 包装器（尽管也可以轻易使用），如下所示：

```
from urllib.request import urlretrieve
from urllib.request import urlopen
from bs4 import BeautifulSoup
import subprocess
import requests
from PIL import Image
from PIL import ImageOps

def cleanImage(imagePath):
    image = Image.open(imagePath)
    image = image.point(lambda x: 0 if x<143 else 255)
    borderImage = ImageOps.expand(image,border=20,fill='white')
    borderImage.save(imagePath)

html = urlopen('http://www.pythonscraping.com/humans-only')
bs = BeautifulSoup(html, 'html.parser')
# 收集需要处理的表单数据（包括验证码和输入字段）
imageLocation = bs.find('img', {'title': 'Image CAPTCHA'})['src']
formBuildId = bs.find('input', {'name':'form_build_id'})['value']
captchaSid = bs.find('input', {'name':'captcha_sid'})['value']
captchaToken = bs.find('input', {'name':'captcha_token'})['value']

captchaUrl = 'http://pythonscraping.com'+imageLocation
urlretrieve(captchaUrl, 'captcha.jpg')
cleanImage('captcha.jpg')
p = subprocess.Popen(['tesseract', 'captcha.jpg', 'captcha'], stdout=
    subprocess.PIPE,stderr=subprocess.PIPE)
p.wait()
f = open('captcha.txt', 'r')

# 清理识别结果中的空白字符
captchaResponse = f.read().replace(' ', '').replace('\n', '')
print('Captcha solution attempt: '+captchaResponse)

if len(captchaResponse) == 5:
    params = {'captcha_token':captchaToken, 'captcha_sid':captchaSid,
              'form_id':'comment_node_page_form', 'form_build_id': formBuildId,
              'captcha_response':captchaResponse, 'name':'Ryan Mitchell',
              'subject': 'I come to seek the Grail',
              'comment_body[und][0][value]':
               '...and I am definitely not a bot'}
    r = requests.post('http://www.pythonscraping.com/comment/reply/10',
                      data=params)
    responseObj = BeautifulSoup(r.text, 'html.parser')
    if responseObj.find('div', {'class':'messages'}) is not None:
        print(responseObj.find('div', {'class':'messages'}).get_text())
else:
    print('There was a problem reading the CAPTCHA correctly!')
```

值得注意的是，有两种异常情况会导致这个程序运行失败。第一种情况是，Tesseract 从验证码图片中识别的结果不是 5 个字符（因为训练样本中验证码的所有有效答案都必须是 5 个字符），结果不会被提交，程序失败。第二种情况是虽然识别的结果是 5 个字符，被提交到了表单，但是服务器对结果不认可，程序仍然失败。在实际运行过程中，第一种情况发生的概率大约为 50%，发生时程序不会向表单提交，程序直接结束并提示验证码识别错误。第二种异常情况发生的概率约为 20%，5 个字符都对的概率约是 30%（每个字符的识别正确率大约是 80%，5 个字符都识别正确的总概率是 32.8%）。

虽然这个程序的识别效果好像很差，但是用户尝试填写验证码的次数并没有限制，而且大多数错误的识别结果都可以在提交到表单之前就被拦下来。因此，如果有一个识别结果提交到表单并传送到服务器，那么验证码很可能就是正确的。如果这样解释并不能让你信服，请记住这些都只是简单的猜测，准确率只有 0.0000001%。[4] 程序只要运行三到四次就可以识别出一个验证码，比简单的猜测 9 亿次还是要节省很多时间的！

注 4：验证码的字符集是 26 个大写字母、26 个小写字母和 10 个数字，5 个字符一共有 62 的 5 次方，即 916 132 832 种可能，因此简单猜测的准确率只有 0.0000001%，下一句中的 9 亿次就是这个道理。
——译者注

第 14 章

避开抓取陷阱

抓取网站的时候，数据显示在浏览器上却抓取不出来；向服务器提交自认为已经处理得很好的表单却被拒绝；自己的 IP 地址不知道什么原因被网站封杀，无法继续访问。没有什么比这些更令人沮丧的了。

这是由于一些堪称最复杂的 bug 还没有解决，不仅因为这些 bug 让人意想不到（程序在一个网站上可以正常使用，但在另一个看起来完全一样的网站上却用不了），还因为那些网站有意不让爬虫抓取信息。网站已经把你定性为一个网络机器人直接拒绝，你无法找出原因。

在这本书里，我已经写了很多方法来处理网站抓取的难点（提交表单，抽取和清理数据，执行 JavaScript，等等）。这一章将继续介绍更多的知识点，尽管属于不同的主题（HTTP header、CSS 和 HTML 表单等），但它们的共同目的都是克服网站阻止自动抓取这个障碍。

即使你觉得下面这些内容现在对你没什么用，我还是强烈建议你至少浏览一下。也许有一天，这一章的内容会帮你解决一个非常复杂的 bug，或者防止该类 bug 出现。

14.1　道德规范

在前几章，我介绍过网页抓取行为在法律上的灰色地带，以及网页抓取涉及的一些道德规范。说实话，从道德角度上说，这一章是我在写这本书时感到最难写的一章。我自己的网站已经被网络机器人、垃圾邮件生成器、网络爬虫和其他各种不受欢迎的虚拟访问者骚扰过很多次了，你的网站可能也是一样。既然如此，我为什么还要在这一章教人们建立更强大的网络机器人呢？

有几个很重要的理由促使我决定写这一章。

- 在抓取那些不想被抓取的网站时，其实存在一些完全符合道德和法律规范的理由。比如我之前的工作就是开发网络爬虫，我曾开发过一个自动信息收集器，在未经许可的情况下，在网站上自动收集客户的名称、地址、电话号码和其他个人信息，然后把抓取的信息提交到网站上，让服务器删除这些客户信息。为了避免竞争，这些网站都会对网络爬虫严防死守。但是，我的工作是确保公司的客户都匿名（其中一些人被跟踪、是家庭暴力受害者，或者因其他正当理由想保持低调），这是进行网页抓取的一个充分的理由，我很高兴自己有能力从事这项工作。
- 虽然不太可能建立一个完全"防爬虫"的网站（最起码得让合法的用户可以方便地访问的网站），但我还是希望本章的内容可以帮助人们保护自己的网站不被恶意攻击。在这一章，我将指出每一种网页抓取技术的缺点，你可以借此保护自己的网站。其实，大多数网络机器人一开始都只能做一些宽泛的信息和漏洞扫描，用本章介绍的几个简单技术就可以挡住 99% 的机器人。但是，它们进化的速度非常快，最好时刻准备迎接新的攻击。
- 和大多数程序员一样，我不认为禁止某一类信息的传播是件百利而无一害的事。

学习这一章的内容时，希望你牢记这里演示的许多程序和介绍的技术都不应该在任何一个网站上使用。不仅因为这么做不好，而且你也可能会收到一封勒令停止警告信，甚至有可能发生更糟糕的事情（关于收到警告信应该怎么办，请参见第 18 章）。不过我不想每次学习新技术时都警告你一下。所以，对于本章后面的内容，如哲学家阿甘曾说的，"我想说的就是这些"。

14.2 让网络机器人看着像人类用户

网站防抓取的前提就是要正确地区分人类用户和网络机器人。虽然网站可以使用很多识别技术（比如验证码）来防止爬虫，但是有些十分简单的方法可以让你的网络机器人看起来更像人类用户。

14.2.1 修改请求头

本书中，我们一直用 Requests 库创建、发送和接收 HTTP 请求，比如在第 10 章中处理网站的表单。Requests 库还是一个设置请求头的利器。HTTP 的请求头是你每次向 Web 服务器发送请求时，传递的一组属性或配置信息。HTTP 定义了几十种古怪的请求头类型，不过大多数都不常用。只有下面 7 个字段被大多数浏览器用来初始化所有网络请求（表中信息是我自己的浏览器数据）。

`Host`	https://www.google.com/
`Connection`	keep-alive

`Accept`	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
`User-Agent`	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36
`Referrer`	https://www.google.com/
`Accept-Encoding`	gzip,deflate,sdch
`Accept-Language`	en-US,en;q=0.8

经典的 Python 爬虫在使用 `urllib` 标准库时，都会发送如下的请求头：

`Accept-Encoding`	identity
`User-Agent`	Python-urllib/3.4

如果你是一个防范爬虫的网站管理员，你会让哪个请求头访问你的网站呢？

幸运的是，请求头是可以用 Requests 库配置的。网站 https://www.whatismybrowser.com 可以用来测试对服务器可见的浏览器属性。你可以用以下代码抓取该网站并验证你的 cookie 设置：

```
import requests
from bs4 import BeautifulSoup

session = requests.Session()
headers = {'User-Agent':'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5)'
           'AppleWebKit 537.36 (KHTML, like Gecko) Chrome',
           'Accept':'text/html,application/xhtml+xml,application/xml;'
           'q=0.9,image/webp,*/*;q=0.8'}
url = 'https://www.whatismybrowser.com/'\
    'detect/what-http-headers-is-my-browser-sending'
req = session.get(url, headers=headers)

bs = BeautifulSoup(req.text, 'html.parser')
print(bs.find('table', {'class':'table-striped'}).get_text)
```

程序输出结果中的请求头应该和程序中设置的 `headers` 是一样的。

虽然网站可能会对 HTTP 请求头的每个属性进行“是否具有人性”的检查，但是我发现通常真正重要的参数就是 `User-Agent`。无论你在做什么项目，一定要记得把 `User-Agent` 属性设置成不容易引起怀疑的内容，不要用 `Python-urllib/3.4`。另外，如果你正在处理一个警觉性非常高的网站，就要注意那些经常用却很少检查的请求头，比如 `Accept-Language` 属性，也许它正是那个网站判断你是个人类访问者的关键。

> **请求头会改变你看世界的方式**
>
> 假设你想为一个研究项目编写一个机器学习语言翻译机，却没有大量的翻译文本来测试它的效果。很多大型网站都会为同样的内容提供不同的语言翻译，并且根据请求头的参数响应不同的语言版本。因此，你只需把请求头属性从 `Accept-Language:en-US` 修改成 `Accept-Language:fr`，就可以从网站上获得“Bonjour”（法语“你好”）这些数据来改善翻译机的翻译效果了（大型跨国企业通常都是很好的抓取对象）。
>
> 请求头还可以让网站改变内容的布局样式。例如，用移动设备浏览网站时，通常会看到一个没有广告、Flash 以及其他干扰的简化的网站版本。因此，把你的请求头 `User-Agent` 改成下面这样，就可以看到一个更容易抓取的网站了！
>
> ```
> User-Agent:Mozilla/5.0 (iPhone; CPU iPhone OS 7_1_2 like Mac OS X)
> AppleWebKit/537.51.2 (KHTML, like Gecko) Version/7.0 Mobile/11D257
> Safari/9537.53
> ```

14.2.2　用JavaScript处理cookie

虽然 cookie 是一把双刃剑，但正确地处理 cookie 可以避免许多抓取问题。网站会用 cookie 跟踪你的访问过程，如果发现了行为异常的爬虫就会中断它的访问，比如特别快速地填写表单，或者浏览大量页面。虽然这些行为可以通过关闭并重新连接网站或者改变 IP 地址来伪装（更多信息请参见第 17 章），但是如果 cookie 暴露了你的身份，再多努力也是白费。

在抓取一些网站时 cookie 是不可或缺的。第 10 章介绍过，在一个网站上持续地保持登录状态，需要你在多个页面中保存一个 cookie。一些网站不要求你在每次登录时都获得一个新 cookie，只要保存一个旧的“已登录”cookie 就可以访问网站。

如果你在抓取一个或者几个目标网站，我建议你检查一下这些网站生成的 cookie，然后想想哪一个 cookie 是爬虫需要处理的。有些浏览器插件可以显示在你访问网站和浏览网站时 cookie 是如何设置的。EditThisCookie 是我最喜欢的 Chrome 浏览器插件之一。

关于使用 Requests 模块处理 cookie 的更多信息，请查看 10.5 节中的示例代码。当然，因为 Requests 库不能执行 JavaScript，所以它不能处理很多由现代跟踪软件（比如 Google Analytics）生成的 cookie，只有当客户端脚本执行后才设置 cookie（或者在用户浏览页面时基于点击按钮等网页事件产生 cookie）。为了处理这些动作，你需要用 Selenium 和 PhantomJS 包（安装方法和基本用法在第 11 章已经介绍过）。

你可以对任意网站（本例用的是 http://pythonscraping.com）调用 webdriver 的 `get_cookie()` 方法来查看 cookie：

```
from selenium import webdriver
driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get('http://pythonscraping.com')
```

```
driver.implicitly_wait(1)
print(driver.get_cookies())
```

这样就可以获得一个非常典型的 Google Analytics 的 cookie 列表：

```
[{'value': '1', 'httponly': False, 'name': '_gat', 'path': '/', 'expi
ry': 1422806785, 'expires': 'Sun, 01 Feb 2015 16:06:25 GMT', 'secure'
: False, 'domain': '.pythonscraping.com'}, {'value': 'GA1.2.161952506
2.1422806186', 'httponly': False, 'name': '_ga', 'path': '/', 'expiry
': 1485878185, 'expires': 'Tue, 31 Jan 2017 15:56:25 GMT', 'secure':
False, 'domain': '.pythonscraping.com'}, {'value': '1', 'httponly': F
alse, 'name': 'has_js', 'path': '/', 'expiry': 1485878185, 'expires':
'Tue, 31 Jan 2017 15:56:25 GMT', 'secure': False, 'domain': 'pythons
craping.com'}]
```

你还可以调用 `delete_cookie()`、`add_cookie()` 和 `delete_all_cookies()` 方法来处理 cookie。另外，还可以保存 cookie 以备其他网络爬虫使用。下面的例子演示了如何把这些函数组合在一起：

```
from selenium import webdriver

phantomPath = '<Path to Phantom JS>'
driver = webdriver.PhantomJS(executable_path=phantomPath)
driver.get('http://pythonscraping.com')
driver.implicitly_wait(1)

savedCookies = driver.get_cookies()
print(savedCookies)

driver2 = webdriver.PhantomJS(executable_path=phantomPath)
driver2.get('http://pythonscraping.com')
driver2.delete_all_cookies()
for cookie in savedCookies:
    if not cookie['domain'].startswith('.'):
        cookie['domain'] = '.{}'.format(cookie['domain'])
    driver2.add_cookie(cookie)

driver2.get('http://pythonscraping.com')
driver.implicitly_wait(1)
print(driver2.get_cookies())
```

在这个例子中，第一个 webdriver 获得了一个网站，打印 cookie 并把它们保存到变量 `savedCookies` 中。第二个 webdriver 加载同一个网站，删除所有的 cookie，然后替换成第一个 webdriver 得到的 cookie。两条技术提示如下所示。

- 第二个 webdriver 在添加 cookie 之前必须加载网站，这样 Selenium 才能知道 cookie 属于哪个域名，尽管加载网站对爬虫没有任何用处。
- 在加载每个 cookie 之前都需要做检查，查看域名是不是以点号（.）字符开头的。这是 PhantomJS 的规则——添加 cookie 的所有域名都要以 . 字符开头(例如，`.pythonscraping.com`)，尽管并不是 PhantomJS webdriver 中的所有 cookie 都遵循这条规则。如果你在使用其他的浏览器 driver，例如 Chrome 或者 Firefox，那么就不需要这么做。

当再次加载这个页面时，两组 cookie 的时间戳、源代码和其他信息应该完全一致。从 Google Analytics 角度看，第二个 webdriver 现在和第一个 webdriver 完全一样，它们会被同样的方式跟踪。如果第一个 webdriver 登录，那么第二个也同样登录。

14.2.3 时间就是一切

一些防护措施完备的网站可能会阻止你快速地提交表单，或者快速地与网站进行交互。即使没有这些安全措施，以比普通人快很多的速度从一个网站下载大量信息也可能让自己被网站封杀。

因此，虽然多线程程序可能是一个快速加载页面的好办法——让你在一个线程中处理数据并在另一个线程中加载页面——但是这对编写好的爬虫来说依然是一种糟糕的策略。应该尽量保证页面加载和数据请求最小化。如果可能，尽量在页面访问之间增加几秒钟的间隔，即使你需要增加一行代码：

```
import time

time.sleep(3)
```

无论你是否需要，页面加载之间的额外几秒钟都是免不了的。有很多次，当我从网站抓取数据的时候，每隔几分钟就需要证明自己“不是一个机器人”（手动识别验证码，给爬虫复制粘贴新的 cookie，从而让被访问网站将爬虫当作“人类”对待），但是通常增加一个延时的 `time.sleep` 就可以解决问题，让我不受限制地抓取。

有时候，你要学会以退为进！

14.3 常见表单安全措施

许多像 Litmus 之类的测试工具已经用了很多年了，现在仍用于区分网络爬虫和使用浏览器的人类访问者，这类手段都取得了不同程度的效果。虽然网络机器人下载一些公开发表的文章和博文并不是什么大事，但是如果网络机器人在你的网站上创建了几千个账号并开始向所有用户发送垃圾邮件，就是一个大问题了。Web 表单，尤其是那些用于账号创建和登录的表单，如果被机器人滥用，就会对网站的安全和计算开销造成严重威胁，因此努力限制网站的接入是最符合许多网站所有者的利益的（至少他们这么认为）。

这些集中在表单和登录环节上的反机器人安全措施，对网络爬虫来说是一个很大的挑战。

记住，当为这些表单创建自动化机器人时，你会遇到的安全措施可不止这些。关于处理受保护表单的更多信息，请参考第 13 章中关于验证码和图片处理的内容，以及第 17 章中关于请求头和 IP 地址处理的内容。

14.3.1　隐含输入字段值

在 HTML 表单中，“隐含”字段以让字段的值对浏览器可见，但是对用户不可见（除非看网页源代码）。随着越来越多的网站开始用 cookie 存储和传递状态变量，隐含字段在短暂失宠之后找到了另一个不错的用处：阻止爬虫自动提交表单。

图 14-1 显示的例子是 Facebook 登录页面上的隐含字段。虽然表单里只有 3 个可见字段（用户名、密码和一个提交按钮），但是在源代码里表单会向服务器传送大量的信息。

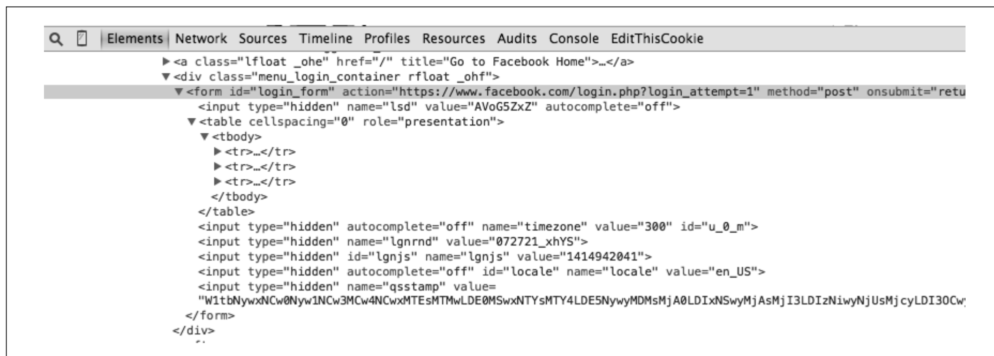


图 14-1：Facebook 登录页面上的隐含字段

用隐含字段阻止网页抓取的方式主要有两种。第一种是表单页面上的一个字段可以用服务器生成的随机变量填充。如果提交时这个值不在表单页面上，服务器就有理由认为它不是从原始表单页面提交的，而是由网络机器人直接提交到表单处理页面的。绕开这个问题的最佳方法是首先抓取表单所在页面上生成的随机变量，然后再提交到表单处理页面。

第二种方式是“蜜罐”（honey pot）。如果表单里包含一个具有普通名称的隐含字段（设置蜜罐圈套），比如“用户名”（username）或“邮箱地址”（email address），设计不太好的网络机器人往往不管这个字段是不是对用户可见，直接填写这个字段并向服务器提交，这样就会中服务器的蜜罐圈套。服务器会忽略所有隐含字段的真实值（或者与表单提交页面的默认值不同的值），而填写隐含字段的用户甚至可能被网站封杀。

总之，有时有必要检查一下表单所在的页面，看看有没有遗漏或弄错一些服务器预先设定好的隐含字段（蜜罐圈套）。如果你看到一些隐含字段，通常带有较大的随机字符串变量，那么 Web 服务器很可能会在表单提交时检查它们。另外，还有其他一些检查，可用来保证当前生成的表单变量只被使用过一次或是最近生成的（这样可以避免变量被简单地存储到一个程序中反复使用）。

14.3.2　避免蜜罐

虽然用 CSS 属性区分有用信息和无用信息很方便（比如，通过读取 `id` 和 `class` 标签获取

信息），但这有时会对网络爬虫造成问题。如果 Web 表单的一个字段通过 CSS 被设置成对用户不可见，那么可以认为普通用户访问网站的时候不能填写这个字段，因为它没有显示在浏览器上。如果这个字段被填写了，就很可能是机器人干的，因此这个提交会失效。

这种手段不仅可以应用在网站的表单上，还可以应用在链接、图片、文件，以及可被机器人读取，但普通用户在浏览器上却看不到的任何内容上面。访问者如果访问了网站上的一个“隐含”链接，就会触发服务器端脚本封杀这个用户的 IP 地址，把这个用户踢出网站，或者采取其他措施禁止这个用户接入网站。实际上，许多商业模式就是在干这些事情。

下面的例子所用的页面是 http://pythonscraping.com/pages/itsatrap.html。这个页面包含了两个链接，其中一个通过 CSS 隐藏了，而另一个是可见的。另外，还有一个包括两个隐含字段的表单：

```
<html>
<head>
    <title>A bot-proof form</title>
</head>
<style>
    body {
        overflow-x:hidden;
    }
    .customHidden {
        position:absolute;
        right:50000px;
    }
</style>
<body>
    <h2>A bot-proof form</h2>
    <a href=
    "http://pythonscraping.com/dontgohere" style="display:none;">Go here!</a>
    <a href="http://pythonscraping.com">Click me!</a>
    <form>
        <input type="hidden" name="phone" value="valueShouldNotBeModified"/><p/>
        <input type="text" name="email" class="customHidden"
                  value="intentionallyBlank"/><p/>
        <input type="text" name="firstName"/><p/>
        <input type="text" name="lastName"/><p/>
        <input type="submit" value="Submit"/><p/>
   </form>
</body>
</html>
```

这 3 个元素通过 3 种不同的方式对用户隐藏：

- 第一个链接通过简单的 CSS 属性设置 `display:none` 进行隐藏；
- 电话号码字段 `name="phone"` 是一个隐含的输入字段；
- 邮箱地址字段 `name="email"` 是通过将元素向右移动 50 000 像素（应该会超出电脑显示器的边界）并隐藏滚动条进行隐藏的。

幸运的是，因为 Selenium 可以获取访问页面的内容，所以它可以区分页面上的可见元素与隐含元素。利用 `is_displayed()` 可以判断元素在页面上是否可见。

例如，下面的代码将获取前面那个页面的内容，然后查找隐含链接和隐含输入字段：

```
from selenium import webdriver
from selenium.webdriver.remote.webelement import WebElement

driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get('http://pythonscraping.com/pages/itsatrap.html')
links = driver.find_elements_by_tag_name('a')
for link in links:
    if not link.is_displayed():
        print('The link {} is a trap'.format(link.get_attribute('href')))

fields = driver.find_elements_by_tag_name('input')
for field in fields:
    if not field.is_displayed():
        print('Do not change value of {}'.format(field.get_attribute('name')))
```

Selenium 抓取出了每个隐含的链接和字段，结果如下所示：

```
The link http://pythonscraping.com/dontgohere is a trap
Do not change value of phone
Do not change value of email
```

虽然你不太可能去访问你找到的那些隐含链接，但是在提交前，记得确认一下那些已经在表单中、准备提交的隐含字段的值（或者让 Selenium 为你自动提交）。总之，简单地忽略隐含字段是很危险的，但与它们交互时一定要小心谨慎。

14.4　问题检查表

这一章（这本书也是一样）的很多内容都是在介绍如何创建一个更像人而不是更像机器人的网络爬虫。如果你一直被网站封杀却找不到原因，这里有个检查表，可以帮你诊断一下问题出在哪里。

- 首先，如果你从 Web 服务器收到的页面是空白的，缺少信息，或者不符合你的预期（或者不是你在浏览器上看到的内容），有可能是因为网站创建页面的 JavaScript 执行有问题。可以看看第 11 章内容。
- 如果你准备向网站提交表单或发出 `POST` 请求，记得检查一下页面的内容，看看你想提交的每个字段是不是都已经填好并且格式正确。用 Chrome 浏览器的检查器之类的工具，查看发送到网站的 `POST` 请求，确认你的每个参数都是正确的。
- 如果你已经登录网站却不能保持登录状态，或者网站上出现了其他的“登录状态”异常，请检查你的 cookie。确保在加载每个页面时 cookie 都被正确调用，而且你的 cookie 在每次发起请求时都发送到了网站上。

- 如果你在客户端遇到了 HTTP 错误，尤其是 403 禁止访问错误，这可能说明网站已经把你的 IP 地址当作机器人了，不再接受你的任何请求。你要么等待你的 IP 地址从网站黑名单里移除，要么就换个 IP 地址（可以去星巴克上网，或者看看第 17 章的内容）。要确保不会再次被封杀，请做到以下几点。
  - 确保你的爬虫在网站上的速度不是特别快。快速抓取是一种糟糕的做法，会对网管的服务器造成沉重的负担，还会让你陷入违法境地，也是 IP 被网站列入黑名单的首要原因。给爬虫增加延迟，让它们在夜深人静的时候运行。切记：匆匆忙忙写程序或收集数据都是拙劣项目管理的表现；应该提前做好计划，避免临阵慌乱。
  - 还有一件必须做的事情：修改你的请求头！有些网站会封杀任何声称自己是爬虫的访问者。如果你不确定请求头的值怎样才算合适，就用你自己浏览器的请求头吧。
  - 确认你没有点击或访问任何人类用户通常不能点击或访问的信息（更多信息请参阅 14.3.2 节）。
  - 如果你用了一大堆复杂的手段才接入网站，考虑联系网管吧，告诉他们你的目的。试试发邮件到 webmaster@< 域名 > 或 admin@< 域名 >，请求网管允许你使用爬虫抓取数据。管理员也是人，你可能会对他们非常配合地分享数据感到惊讶。

第15章

用爬虫测试网站

当研发一个技术栈较大的 Web 项目时，经常只对栈底（项目后期用的技术）定期进行测试。目前大多数编程语言（包括 Python）都拥有某种测试框架，但是网站的前端通常并没有自动化测试，尽管前端才是整个项目中真正与用户零距离接触的唯一一个部分。

部分原因是网站经常混用了许多不同的标记语言和编程语言。你可以为 JavaScript 部分写单元测试，但没什么用，因为如果与 JavaScript 交互的 HTML 内容改变了，那么即使 JavaScript 可以正常地运行，也不能完成网页需要的动作。

网站的前端测试经常最后才做，或者指派给低级程序员去做，最多再给他们一个检查表和一个 bug 跟踪器。但其实只要再稍微努点儿力，我们就可以把检查表变成一系列单元测试，用网络爬虫代替人眼进行测试。

想象有一个由测试驱动的 Web 开发项目。每天都要做测试，以保证网络接口各个部分的功能都正常。每当有新的特性加入网站，或者某个元素的位置发生了改变，就执行一组自动化测试。这一章将介绍测试的基础知识，以及如何用 Python 网络爬虫测试各种简单或复杂的网站。

15.1　测试简介

如果你从没为你的代码写过测试，那么现在开始再合适不过了。运行一套测试来保证你的代码按预期运行，不仅可以节约你的时间，减少你对 bug 的忧虑，还可以让升级变得更加简单。

什么是单元测试

测试和**单元测试**（unit test）这两个词基本可以看成是等价的。通常，当程序员说“写测试”时，他们真正的意思就是“写单元测试”。而一些程序员提到写单元测试时，他们写的就是某一种测试。

虽然不同公司的单元测试定义和实践方法大相径庭，但是单元测试通常具有以下特点。

- 每个单元测试用于测试一个组件的功能的一个方面。例如，如果从银行账户取出金额为负数的一笔款，那么单元测试就要确保抛出适当的错误信息。
  通常，一个组件的所有单元测试都集成在同一个类（class）里。你可能有一个测试是针对从银行账户取出金额为负数的一笔款，另一个是针对透支银行账户行为的单元测试。
- 每个单元测试都可以完全独立地运行，一个单元测试需要的所有启动（setup）和卸载（teardown）都必须通过这个单元测试本身去处理。单元测试不能对其他测试造成干扰，而且不论按何种顺序排列，它们都必须能够正常地运行。
- 每个单元测试通常至少包含一个**断言**（assertion）。例如，一个单元测试可以断言 2+2 等于 4。有时，一个单元测试也许只包含一个失败状态（failure state）。例如，如果抛出异常，则测试失败；如果一切顺利，则测试默认通过。
- 单元测试与生产代码是分离的。虽然它们需要导入并使用待测试的代码，但是它们一般被放在单独的类和目录中。

尽管有很多测试类型可写，比如集成测试和验证测试等，但本章只重点介绍单元测试。这不仅仅是因为单元测试在当前的测试驱动开发中十分主流，还因为其代码长度和灵活性使它们非常适合作为示例。另外，Python 自带单元测试标准库，下一节就来介绍它。

15.2 Python单元测试

所有标准版 Python 安装后都有单元测试模块 `unittest`。只要导入并扩展 `unittest.TestCase` 类，就可以实现下面的功能：

- 为每个单元测试的开始和结束提供 `setUp` 和 `tearDown` 函数
- 提供不同类型的“断言”语句，让测试成功或失败
- 把所有以 `test_` 开头的函数当作单元测试运行，忽略不带 `test_` 的函数

下面的例子演示了如何用 Python 实现一个非常简单的单元测试来测试 2+2=4：

```
import unittest

class TestAddition(unittest.TestCase):
    def setUp(self):
        print('Setting up the test')
```

```
    def tearDown(self):
        print('Tearing down the test')

    def test_twoPlusTwo(self):
        total = 2+2
        self.assertEqual(4, total);

if __name__ == '__main__':
    unittest.main()
```

虽然 setUp 和 tearDown 函数在这里并没有实现有用的功能，但是仍然达到了演示的目的。需要注意的是，这两个函数在每个测试开始和结束时都会运行一次，而不是在类中所有测试开始之前和结束之后各运行一次。

从命令行运行时，测试函数的输出如下：

```
Setting up the test
Tearing down the test
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

这表明测试运行成功，2+2 的确等于 4。

在 Jupyter notebook 中运行单元测试

本章单元测试代码的开头都是如下形式：

```
if __name__ == '__main__':
    unittest.main()
```

仅当 `if __name__ == '__main__'` 这行代码在 Python 中直接运行，而不是通过一个导入语句运行时，这个判断才为真。它可以让你用 unittest.TestCase 类在命令行直接运行你的单元测试。

在 Jupyter notebook 中，情况有点不一样。Jupyter 创建的 argv 参数可能会在单元测试中引起错误，而且由于在测试运行之后，unittest 框架默认会退出 Python（这会导致 notebook 内核发生错误），我们必须阻止这样的现象发生。

在 Jupyter notebook 中，你可以用以下命令启动单元测试：

```
if __name__ == '__main__':
    unittest.main(argv=[''], exit=False)
    %reset
```

第二行将所有的 argv 变量（命令行参数）设置成一个空字符串，它就会被 unnittest.main 忽略。它还阻止了 unittest 在测试运行之后退出。

%reset 行也非常有用，因为它重置了内存，并销毁了所有用户在 Jupyter notebook 中创建的变量。如果没有该语句，你在 notebook 中编写的每个单元测试都将包含此前运行的测试的所有方法，它们也继承了 unittest.TestCase，包括 setUp 和 tearDown 方法。这就意味着每个单元测试将运行此前单元测试的所有方法。

当然，使用 %reset 确实给运行测试的用户额外创建了一个手动步骤。当运行测试时，notebook 将弹出提示，询问用户是否要重置内存。输入 y 后点击回车键就可以了。

测试维基百科

将 Python 的 unittest 库与网络爬虫组合起来，就可以实现网站前端的测试了（除了 JavaScript 测试，后面会介绍）。

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import unittest

class TestWikipedia(unittest.TestCase):
    bs = None
    def setUpClass():
        url = 'http://en.wikipedia.org/wiki/Monty_Python'
        TestWikipedia.bs = BeautifulSoup(urlopen(url), 'html.parser')

    def test_titleText(self):
        pageTitle = TestWikipedia.bs.find('h1').get_text()
        self.assertEqual('Monty Python', pageTitle);

    def test_contentExists(self):
        content = TestWikipedia.bs.find('div',{'id':'mw-content-text'})
        self.assertIsNotNone(content)

if __name__ == '__main__':
    unittest.main()
```

这里有两个测试：第一个测试页面的标题是否为“Monty Python”，另一个测试页面是否有一个 div 节点的 id 属性是 "mw-content-text"。

需要注意的是，这个页面的内容只加载一次，全局对象 bs 由多个测试共享。这是通过 unittest 类的函数 setUpClass 来实现的，这个函数只在类的初始化阶段运行一次（与每个测试启动时都运行的 setUp 函数不同）。用 setUpClass 代替 setUp 可以省去不必要的页面加载；我们可以一次性抓取全部内容，供多个测试使用。

除了运行时间和频次不同之外，setUpClass 和 setUp 的一个主要架构区别是：setUpClass 是一个静态方法，它属于类本身并且拥有全局类变量，而 setUp 是一个实例函数，它属于类的一个特定实例。这就是为什么 setUp 可以设置自身的属性（即这个类的特定实例），而 setUpClass 只能获取 TestWikipedia 类的静态类属性。

虽然一次只测试一个页面可能不够强大，也没什么意思，但是如第 3 章所述，你可以轻松地创建一个网络爬虫去遍历网站中所有的页面。下面我们来看看，当把网络爬虫和一个向页面内容添加断言的单元测试组合起来时，会发生什么。

重复执行一个测试的方法有多种，但是针对要在页面上运行的每组测试，每个页面必须只加载一次，而且你必须避免在内存中一次性加入大量的信息。具体设置如下所示：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import unittest
import re
import random
from urllib.parse import unquote

class TestWikipedia(unittest.TestCase):

    def test_PageProperties(self):
        self.url = 'http://en.wikipedia.org/wiki/Monty_Python'
        # 测试遇到的前10个页面
        for i in range(1, 10):
            self.bs = BeautifulSoup(urlopen(self.url), 'html.parser')
            titles = self.titleMatchesURL()
            self.assertEquals(titles[0], titles[1])
            self.assertTrue(self.contentExists())
            self.url = self.getNextLink()
        print('Done!')

    def titleMatchesURL(self):
        pageTitle = self.bs.find('h1').get_text()
        urlTitle = self.url[(self.url.index('/wiki/')+6):]
        urlTitle = urlTitle.replace('_', ' ')
        urlTitle = unquote(urlTitle)
        return [pageTitle.lower(), urlTitle.lower()]

    def contentExists(self):
        content = self.bs.find('div',{'id':'mw-content-text'})
        if content is not None:
            return True
        return False

    def getNextLink(self):
        # 利用第3章中介绍的技术返回页面上的随机链接
        links = self.bs.find('div', {'id':'bodyContent'}).find_all(
            'a', href=re.compile('^(/wiki/)((?!:).)*$'))
        randomLink = random.SystemRandom().choice(links)
        return 'https://wikipedia.org{}'.format(randomLink.attrs['href'])

if __name__ == '__main__':
    unittest.main()
```

有几个地方需要注意。首先，这个类里面实际上只有一个测试。其他的函数其实都是辅助函数（helper function），即使它们做了大量的计算来判断测试是否通过。因为测试函数

`test_PageProperties` 使用了断言语句，所以测试的结果最终会传到这些断言所在的测试函数里。

另外，`contentExists` 返回的是布尔变量，`titleMatchesURL` 返回的是字符串列表。为什么要传递列表而不是布尔变量呢？将如下布尔型断言的结果：

```
======================================================================
FAIL: test_PageProperties (__main__.TestWikipedia)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "15-3.py", line 22, in test_PageProperties
    self.assertTrue(self.titleMatchesURL())
AssertionError: False is not true
```

与 `assertEquals` 语句的结果相比：

```
======================================================================
FAIL: test_PageProperties (__main__.TestWikipedia)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "15-3.py", line 23, in test_PageProperties
    self.assertEquals(titles[0], titles[1])
AssertionError: 'lockheed u-2' != 'u-2 spy plane'
```

究竟哪一种方式调试起来更方便呢？（在这个示例中，之所以会发生错误，是因为网页发生了重定向，即词条 https://en.wikipedia.org/wiki/U-2_spy_plane 跳转到了标题为“Lockheed U-2”的词条。）

15.3 Selenium单元测试

和第 11 章介绍的 Ajax 抓取一样，在网站测试中 JavaScript 也是一个难题。幸运的是，我们有 Selenium，它是一个可以解决网站上各种复杂问题的优秀的测试框架；其实，它的设计初衷就是用来做网站测试！

虽然这里的单元测试都是同一种语言（Python）写的，但是 Python 单元测试和 Selenium 单元测试的语法还是有很大不同。Selenium 不要求单元测试必须是类的一个函数，它的“断言”语句也不需要括号，而且测试通过时不会有提示，只有当测试失败时才会给出提示。

```
driver = webdriver.PhantomJS()
driver.get('http://en.wikipedia.org/wiki/Monty_Python')
assert 'Monty Python' in driver.title
driver.close()
```

当这个测试运行的时候，不会输出任何信息。

因此，Selenium 单元测试可以比 Python 单元测试写得更加随意，而断言语句甚至可以整合到生产代码中，非常适合某个条件不能满足就中断代码执行的需求。

与网站进行交互

最近，我想通过本地一个小商家网站上的联系方式表单联系该商家，结果发现表单出问题了，我点击提交按钮的时候没有反应。经过一番探索之后，我发现这个网站用了一个简易的邮件发送表单，如果商户联系方式的内容有问题就可以给网管发邮件。于是我就用这个邮箱地址给他们发了一封邮件，告诉他们联系方式表单出了问题，让他们尽快解决，虽然不是技术问题。

如果我写一个普通的爬虫来抓取或测试这个表单，那么爬虫也许只能复制表单的结构，然后直接给我自己发邮件，不过抓不到表单的内容。那么我怎么测试表单的功能才能保证它在浏览器上也可以正常工作呢？

虽然在前面几章中我们介绍过链接跳转、表单提交和其他网站交互行为，但是我们做那些事情的共同初衷都是要**避开**浏览器图形界面，而不是使用浏览器。另一方面，Selenium 可以在浏览器（这里用 PhantomJS 无头浏览器）中做任何事，包括输入文字、点击按钮等，这样就可以找出异常表单、JavaScript 代码错误、HTML 排版错误，以及在用户使用过程中可能出现的其他问题。

这个测试的关键是使用 Selenium 的 `elements`。这个对象在第 11 章已经简单介绍过了，它的调用方式如下所示：

```
usernameField = driver.find_element_by_name('username')
```

就像你可以在浏览器里对网站上的不同元素执行一系列操作一样，Selenium 也可以对任何给定元素执行很多操作，如下所示：

```
myElement.click()
myElement.click_and_hold()
myElement.release()
myElement.double_click()
myElement.send_keys_to_element('content to enter')
```

除了一次性完成一个元素的多个操作，还可以将一组操作组合成一个**动作链**（action chain）存储起来，然后在一个程序中执行一次或多次。动作链可以方便地组合多个操作，非常有用，而且其功能和前面示例中对一个元素显式调用操作是完全一样的。

为了演示两种方式的差异，我们看一看 http://pythonscraping.com/pages/files/form.html 的表单（是第 10 章用过的例子）。我们用下面的方式填写表单并提交：

```
from selenium import webdriver
from selenium.webdriver.remote.webelement import WebElement
from selenium.webdriver.common.keys import Keys
from selenium.webdriver import ActionChains


driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get('http://pythonscraping.com/pages/files/form.html')

firstnameField = driver.find_element_by_name('firstname')
lastnameField = driver.find_element_by_name('lastname')
submitButton = driver.find_element_by_id('submit')

### 方法1 ###
firstnameField.send_keys('Ryan')
lastnameField.send_keys('Mitchell')
submitButton.click()
################

### 方法2 ###
actions = ActionChains(driver).click(firstnameField).send_keys('Ryan')
                              .click(lastnameField).send_keys('Mitchell')
                              .send_keys(Keys.RETURN)
actions.perform()

################

print(driver.find_element_by_tag_name('body').text)

driver.close()
```

方法 1 在两个字段上调用 `send_keys`，然后点击“提交”按钮；而方法 2 用一个动作链来点击每个字段并填写内容，这些行为是在 `perform` 调用之后才发生的。无论用第一个方法还是第二个方法，这个程序的结果都一样：

```
Hello there, Ryan Mitchell!
```

除了用来处理命令的对象不同之外，这两个方法还有一个差异：注意第一个方法提交表单时点击的是“提交”按钮，而第二个方法提交表单时用的是回车键（`Keys.RETURN`）。因为实现同样效果的事件发生顺序可以有多种，所以用 Selenium 实现同样的结果也有许多方式。

1. 鼠标拖放动作

单击按钮和输入文字只是 Selenium 的一个功能，其真正的亮点是能够处理新形式的 Web 交互。Selenium 可以轻松地完成鼠标拖放动作。使用它的拖放功能，你需要指定一个被拖放的元素以及拖放的距离或者拖放到的目标元素。

下面的例子用 http://pythonscraping.com/pages/javascript/draggableDemo.html 页面演示了拖放动作：

```
from selenium import webdriver
from selenium.webdriver.remote.webelement import WebElement
from selenium.webdriver import ActionChains

driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get('http://pythonscraping.com/pages/javascript/draggableDemo.html')

print(driver.find_element_by_id('message').text)

element = driver.find_element_by_id('draggable')
target = driver.find_element_by_id('div2')
actions = ActionChains(driver)
actions.drag_and_drop(element, target).perform()

print(driver.find_element_by_id('message').text)
```

示例页面的 message 节点上显示了两条信息。第一条是：

```
Prove you are not a bot, by dragging the square from the blue area to the red
area!
```

然后任务很快就会完成，第二条内容就被打印出来：

```
You are definitely not a bot!
```

就像示例页面上显示的，很多验证码都使用拖动来证明访问者不是机器人。虽然机器人也可以长时间拖着一个元素不放（就是点击，按住，移动），但是也不知道为什么，用“拖动”来检验一个用户是不是机器人的方式仍然存在。

另外，这些可拖放的验证码库很少使用那些能够难住机器人的任务，比如“拖动小猫图片，放到奶牛图片的上面”（这需要你能够识别“小猫”和“奶牛”图片，同时解析指令）；相反，它们经常用数字排序或其他一些非常简单的任务，就像前面例子里的拖放。

当然，这些验证码库的优势在于那些简单任务可以实现大量的变化，而且每种变化的使用频率都不高，另外也不会有人愿意花时间去开发一个能够搞定所有任务的机器人。这个例子可以解释为什么你不应该在大型网站上使用这种技术。

2. 截屏

除了普通的测试功能，Selenium 还有一个有趣的技巧可以让你的测试更加容易（或者让你的老板更喜欢）：截屏。截屏可以在单元测试中创建，而无须点击截屏按钮：

```
driver = webdriver.PhantomJS()
driver.get('http://www.pythonscraping.com/')
driver.get_screenshot_as_file('tmp/pythonscraping.png')
```

这段脚本会访问 http://pythonscraping.com/，并将主页的屏幕截图保存在本地的 tmp 文件夹中（该文件夹必须已创建好，以供正确存储之用）。截屏可保存为多种文件格式。

15.4　单元测试与Selenium单元测试的选择

Python 单元测试的语法严谨且冗长，更适合为大多数大型项目写测试，而 Selenium 的测试方式灵活且功能强大，可以成为一些网站功能测试的首选。那么应该使用哪个呢？

答案是：不需要选择。Selenium 可以轻易地获取网站的信息，而单元测试可以评估这些信息是否满足通过测试的条件。因此，你没有理由拒绝把 Selenium 导入 Python 的单元测试，两者组合是最佳拍档。

例如，下面的程序创建了一个带拖放动作的网站单元测试，如果一个元素被正确地拖放到另一个元素里，那么推断条件成立，会显示“你不是一个机器人！”（You are not a bot!），测试通过。

```
from selenium import webdriver
from selenium.webdriver.remote.webelement import WebElement
from selenium.webdriver import ActionChains
import unittest

class TestDragAndDrop(unittest.TestCase):
    driver = None
    def setUp(self):
        self.driver = webdriver.PhantomJS(executable_path='<Path to PhantomJS>')
        url = 'http://pythonscraping.com/pages/javascript/draggableDemo.html'
        self.driver.get(url)

    def tearDown(self):
        print("Tearing down the test")

    def test_drag(self):
        element = self.driver.find_element_by_id('draggable')
        target = self.driver.find_element_by_id('div2')
        actions = ActionChains(self.driver)
        actions.drag_and_drop(element, target).perform()
        self.assertEqual('You are definitely not a bot!',
            self.driver.find_element_by_id('message').text)

if __name__ == '__main__':
    unittest.main(argv=[''], exit=False)
```

基本上，网站上的任何内容都可以用 Python 单元测试和 Selenium 的组合来测试。其实，如果再与第 13 章介绍的一些图像处理库结合起来，就可以通过网站截屏实现像素级测试了！

第 16 章

并行网页抓取

网页抓取的速度很快，最起码通常比雇用十几个实习生手动从网上复制数据要快很多。当然，随着技术的不断进步和享乐适应，人们在某个时刻会觉得这还是“不够快”。于是人们开始把目光转向分布式计算。

和其他技术领域不同，网页抓取通常并不能单纯依靠“给问题增加更多的进程”来提升速度。虽然运行一个进程（process）很快，但是运行两个进程未必能将速度提升一倍。而当运行 3 个进程时，可能你的所有请求都会被远程服务器封杀，因为它认为你是在恶意攻击。

然而，在某些场景中使用并行网页抓取或者并行线程（thread）/ 进程仍然有些好处：

- 从多个数据源（多个远程服务器）而不只是一个数据源收集数据；
- 收集数据的同时，在已收集到的数据上执行时间更长 / 更复杂的操作（例如图像分析或者 OCR 处理）；
- 从大型 Web 服务收集数据，如果你已经付费，或者创建多个连接是使用协议允许的行为。

16.1 进程与线程

Python 既支持多进程（multiprocessing），也支持多线程（multithreading）。多进程和多线程可以实现相同的目标：同时执行两个编程任务，而不是像传统线性方式那样一次只执行一个任务。

在计算机科学中，运行在操作系统中的每个进程都可以拥有多个线程。每个进程具有自己独享的内存，这意味着进程里面的多个线程可以共享同一块内存，而多个进程之间不能共

享内存，而且必须显式地进行通信。

用多线程编程执行任务时，多个线程可以共享内存，因此通常认为这比多进程编程更简单。但是，这种便利也需要付出代价。

Python 的全局解释器锁（global interpreter lock，GIL）会阻止多个线程同时运行同一行代码。GIL 确保由所有进程共享的内存不会中断（例如，内存中的字节用一个值写一半，用另一个值写另一半）。虽然这个锁可以让你写多线程的程序，并在同一时刻获取代码的运行结果，但是这么做存在性能瓶颈。

16.2　多线程抓取

Python 3.x 版本的用户请使用 `_thread` 模块，`thread` 模块已经被废弃。

下面的示例展示了用多个线程来实现一个任务：

```
import _thread
import time

def print_time(threadName, delay, iterations):
    start = int(time.time())
    for i in range(0,iterations):
        time.sleep(delay)
        seconds_elapsed = str(int(time.time()) - start)
        print ("{} {}".format(seconds_elapsed, threadName))

try:
    _thread.start_new_thread(print_time, ('Fizz', 3, 33))
    _thread.start_new_thread(print_time, ('Buzz', 5, 20))
    _thread.start_new_thread(print_time, ('Counter', 1, 100))
except:
    print ('Error: unable to start thread')

while 1:
    pass
```

这参考了经典的 FizzBuzz 编程测试，运行代码会得到一堆冗长的结果：

```
1 Counter
2 Counter
3 Fizz
3 Counter
4 Counter
5 Buzz
5 Counter
6 Fizz
6 Counter
```

这个脚本开启了 3 个线程：一个线程每 3 秒打印一次“Fizz”，另一个线程每 5 秒打印一次

“Buzz”，第三个线程每秒打印一次“Counter”。

当 3 个线程启动之后，程序主线程首先命中 `while 1` 循环语句，让程序（及其子线程）一直运行，直到用户输入 Ctrl-C 才会终止。

除了打印“Fizz”和“Buzz”，你还可以用多线程实现一个有用的任务，例如抓取一个网站：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import random

import _thread
import time

def get_links(thread_name, bs):
    print('Getting links in {}'.format(thread_name))
    return bs.find('div', {'id':'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))

# 为线程定义一个函数
def scrape_article(thread_name, path):
    html = urlopen('http://en.wikipedia.org{}'.format(path))
    time.sleep(5)
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    print('Scraping {} in thread {}'.format(title, thread_name))
    links = get_links(thread_name, bs)
    if len(links) > 0:
        newArticle = links[random.randint(0, len(links)-1)].attrs['href']
        print(newArticle)
        scrape_article(thread_name, newArticle)


# 创建两个线程
try:
    _thread.start_new_thread(scrape_article, ('Thread 1', '/wiki/Kevin_Bacon',))
    _thread.start_new_thread(scrape_article, ('Thread 2', '/wiki/Monty_Python',))
except:
    print ('Error: unable to start threads')

while 1:
    pass
```

请注意函数里的这行代码：

```
time.sleep(5)
```

因为你现在抓取维基百科的速度几乎是使用单线程时的两倍，所以这行代码可以防止脚本给维基百科服务器增加太多负载。其实，在请求数量不是问题的服务器上运行脚本时，这行代码可以省略。

如果你想简单重写一下代码，从而跟踪两个线程已经看到的相同文章，以便没有文章被访问过两次，该怎么办呢？你可以在多线程环境中使用列表，就像在单线程环境中使用一样：

```
visited = []
def get_links(thread_name, bs):
    print('Getting links in {}'.format(thread_name))
    links = bs.find('div', {'id':'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))
    return [link for link in links if link not in visited]

def scrape_article(thread_name, path):
    visited.append(path)
```

需要注意的是，现在 `scrape_article` 函数的第一个动作，是把当前网页的路径添加到已经浏览过的路径列表中。这会减小抓取两次的可能性，但是不会彻底解决这类问题。

如果你运气不够好，可能两个线程还是会同时抓取同一个路径，它们都发现该路径不在浏览过的路径列表中，然后同时将它加入列表，并同时抓取内容。不过，实践中这类情况极少发生，因为爬虫运行的速度不一致，而且维基百科包含的页面数量巨大[1]。

这就是**竞争条件**（race condition）的一个例子。由于竞争条件难以调试，甚至经验丰富的程序员也搞不定，因此，评估这些代码的潜在风险，估计其可能性，并预测其影响的严重性是非常重要的。

在这个示例的竞争条件中，爬虫在同一页面上抓取过两次，可能没必要写代码去处理。

16.2.1　竞争条件与队列

虽然你可以用列表进行线程间的通信，但是列表不是专门为线程间通信而设计的，误用列表很容易导致程序运行变慢，甚至在竞争条件中产生错误。

虽然列表擅长添加和读取元素，但是移除任意位置的元素时效率并不高，尤其是处理列表头部的元素时。用下面这行代码实际上需要 Python 重写整个列表，这显然会降低程序的运行速度。

```
myList.pop(0)
```

更危险的是，列表让原本只是偶然发生的非线程安全写入变得十分容易。例如：

```
myList[len(myList)-1]
```

这行代码在多线程环境下可能并不能获取列表末尾的元素，另外，如果 `len(myList)-1` 的值是在另一个操作修改列表之前瞬间计算的，这甚至可能引发异常。

注 1：两个线程起点不同，自顾不暇，难以重叠。——译者注

有人可能认为，上面的语句可以用更“符合 Python 风格”的写法 `myList[-1]`。当然，不会有人一时糊涂写出了不符合 Python 风格的代码（尤其是像 Java 程序员习惯的写法 `myList[myList.length-1]`）！但是，即使你的代码写得无可非议，列表也可能出现其他非线程安全的形式：

```
my_list[i] = my_list[i] + 1
my_list.append(my_list[-1])
```

这两种形式都会导致竞争条件，造成意想不到的结果。因此，我们需要抛弃列表，用非列表变量向线程传递信息！

```
# 从全局列表读取信息
my_message = global_message
# 向全局列表写入信息
global_message = 'I've retrieved the message'
# 对信息进行一些处理
```

这似乎很好，直到你意识到，在第一行和第二行语句之间的一瞬，一个线程可能无意中覆盖了来自另一个线程的另一条消息“I’ve retrieved the message”（我收到了你的消息）。现在，你只需要为每个线程构建一系列详细的个人消息对象，里面包含一些确定哪个线程得到了什么消息的逻辑……或者也可以使用专门为此目的而创建的 `Queue` 模块。

队列是一种类似于列表的对象，有先进先出（First In First Out，FIFO）方法，也有后进先出（Last In First Out，LIFO）方法。队列通过 `queue.put('My message')` 从任意线程接收数据，然后再给调用 `queue.get()` 方法的线程发送数据。

队列并不是设计用来存储静态数据的，而是用来以线程安全的方式传送静态数据的。从队列中检索出来之后，数据应该只存在于检索它的线程中。因此，队列经常用于委托任务或者发送临时通知。

这个特征在网页抓取中非常有用。例如，假设你想将爬虫收集的数据保存到数据库中，并且想让每个线程都能够快速保存数据。虽然所有线程用一个共享数据库连接可能会出现问题（单个数据库连接不能并行处理请求），但是给每个抓取线程单独配置一个数据库连接又没什么意义。随着爬虫的规模不断增大（你可能会用几百个不同的线程从一百个网站收集数据），可能会出现大量的空闲数据库连接，只是在一个页面加载后偶尔写入一次数据。

相反，你可以采用较少的数据库线程，每个线程都有独立的连接，从队列来回获取并存储数据。这样可以实现更加可控的数据库连接。

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import random
import _thread
```

```
from queue import Queue
import time
import pymysql

def storage(queue):
    conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
        user='root', passwd='', db='mysql', charset='utf8')
    cur = conn.cursor()
    cur.execute('USE wiki_threads')
    while 1:
        if not queue.empty():
            article = queue.get()
            cur.execute('SELECT * FROM pages WHERE path = %s',
                (article["path"]))
            if cur.rowcount == 0:
                print("Storing article {}".format(article["title"]))
                cur.execute('INSERT INTO pages (title, path) VALUES (%s, %s)', \
                    (article["title"], article["path"]))
                conn.commit()
            else:
                print("Article already exists: {}".format(article['title']))

visited = []
def getLinks(thread_name, bs):
    print('Getting links in {}'.format(thread_name))
    links = bs.find('div', {'id':'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))
    return [link for link in links if link not in visited]

def scrape_article(thread_name, path, queue):
    visited.append(path)
    html = urlopen('http://en.wikipedia.org{}'.format(path))
    time.sleep(5)
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    print('Added {} for storage in thread {}'.format(title, thread_name))
    queue.put({"title":title, "path":path})
    links = getLinks(thread_name, bs)
    if len(links) > 0:
        newArticle = links[random.randint(0, len(links)-1)].attrs['href']
        scrape_article(thread_name, newArticle, queue)

queue = Queue()
try:
    _thread.start_new_thread(scrape_article, ('Thread 1',
        '/wiki/Kevin_Bacon', queue,))
    _thread.start_new_thread(scrape_article, ('Thread 2',
        '/wiki/Monty_Python', queue,))
    _thread.start_new_thread(storage, (queue,))
except:
    print ('Error: unable to start threads')

while 1:
    pass
```

这个脚本创建了 3 个线程：两个线程用随机方式从维基百科抓取网页，第三个线程在 MySQL 数据库中存储数据。关于 MySQL 和数据存储的更多信息，请参见第 6 章。

16.2.2 threading模块

Python 的 _thread 模块是相当底层的模块，虽然它可以让你对线程进行细致的管理，但是由于它没有提供高级函数，因此你需要事必躬亲，用起来比较费劲。而 threading 模块是一个高级接口，可以让你轻松地使用线程，同时也暴露了 _thread 模块的所有特性。

例如，你可以用 enumerate 之类的静态函数获取所有活跃线程的列表，这些线程通过 threading 模块进行初始化，无须你手动跟踪它们。类似地，函数 activeCount 可以获得总线程数。_thread 的许多函数都换了更方便、更好记的名字，比如获取当前线程名称的 get_ident 就换成了 currentThread。

下面一个简单的线程示例：

```
import threading
import time

def print_time(threadName, delay, iterations):
    start = int(time.time())
    for i in range(0,iterations):
        time.sleep(delay)
        seconds_elapsed = str(int(time.time()) - start)
        print ('{} {}'.format(seconds_elapsed, threadName))

threading.Thread(target=print_time, args=('Fizz', 3, 33)).start()
threading.Thread(target=print_time, args=('Buzz', 5, 20)).start()
threading.Thread(target=print_time, args=('Counter', 1, 100)).start()
```

这个示例会产生和前面 _thread 示例相同的“FizzBuzz”结果。

threading 模块的一个优点是，它可以轻松地创建其他线程都无法访问的线程局部数据（local thread data）。这样做的好处是，如果你有若干线程，它们各自抓取不同的网站，那么每个线程都可以跟踪自己访问的页面列表。

局部数据可以随时创建，调用线程函数 threading.local() 即可：

```
import threading

def crawler(url):
    data = threading.local()
    data.visited = []
    # 抓取网站

threading.Thread(target=crawler, args=('http://brookings.edu')).start()
```

这样就可以解决线程之间因为共享对象而导致竞争条件的问题。无论何时，只要不需要共享对象，就不要共享，保存在线程局部内存中即可。为了安全地在线程中共享对象，仍然可以使用上一节中的 Queue 模块。

threading 模块不但扮演了线程保姆的角色，而且可以对保姆的责任进行高度定制。isAlive 函数的默认行为是查看是否有线程仍然处于活跃状态。只有当一个线程完成抓取（或崩溃）之后，该函数才会返回 True。

通常情况下，爬虫都需要运行很长时间。isAlive 函数可以确保爬虫在一个线程崩溃后重启：

```
threading.Thread(target=crawler)
t.start()

while True:
    time.sleep(1)
    if not t.isAlive():
        t = threading.Thread(target=crawler)
        t.start()
```

其他的监控方法也可以通过扩展 threading.Thread 对象来实现：

```
import threading
import time

class Crawler(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.done = False

    def isDone(self):
        return self.done

    def run(self):
        time.sleep(5)
        self.done = True
        raise Exception('Something bad happened!')

t = Crawler()
t.start()

while True:
    time.sleep(1)
    if t.isDone():
        print('Done')
        break
    if not t.isAlive():
        t = Crawler()
        t.start()
```

示例中新的 Crawler 类包括一个 isDone 方法，可以用来检查爬虫是否已经完成抓取任务。这么做的好处是，如果还有其他的日志方法仍在执行，那么线程就不能关闭，但是抓取工

作其实已经完成了。通常，isDone 也可以用某种状态或者进度条来代替，例如，有多少页面已经记录日志，当前已经抓取到哪一页，等等。

Crawler.run 遇到任何异常都会让 Crawler 类重启，只有当 isDone 返回 True 时，程序才会退出。

在 Crawler 类中对 threading.Thread 进行扩展，不但可以改善爬虫的稳定性和灵活性，还可以让你一次性监控多个爬虫的任意属性。

16.3　多进程抓取

Python 的 Processing 模块可以从程序主进程创建能够被启动（start）和连接（join）的新进程。下面的代码使用了上一节中的 FizzBuzz 示例。

```
from multiprocessing import Process
import time

def print_time(threadName, delay, iterations):
    start = int(time.time())
    for i in range(0,iterations):
        time.sleep(delay)
        seconds_elapsed = str(int(time.time()) - start)
        print (threadName if threadName else seconds_elapsed)


processes = []
processes.append(Process(target=print_time, args=('Counter', 1, 100)))
processes.append(Process(target=print_time, args=('Fizz', 3, 33)))
processes.append(Process(target=print_time, args=('Buzz', 5, 20)))

for p in processes:
    p.start()
for p in processes:
    p.join()
```

操作系统将每一个进程都看作一个独立的程序。如果检查你的操作系统的活动监视器或任务管理器，就会看到类似图 16-1 的情景。

Activity Monitor (My Processes)

CPU　Memory　Energy　Disk　Network　python

Process Name	Bytes Written	Bytes Read	Kind	PID	User
python3.6	0 bytes	0 bytes	64 bit	83561	rmitchell
python3.6	0 bytes	0 bytes	64 bit	83562	rmitchell
python3.6	0 bytes	0 bytes	64 bit	83563	rmitchell
python3.6	4 KB	14.0 MB	64 bit	76154	rmitchell
python3.6	0 bytes	0 bytes	64 bit	83560	rmitchell

图 16-1：在运行 FizzBuzz 示例时，有 5 个 Python 进程同时运行

图中第四个进程 PID 76154 是 Jupyter notebook 实例，如果你运行 iPython notebook 就会出现。第五个进程 83560 是程序主进程，在程序首次运行时启动。操作系统分配的 PID 都是增序的。除非在 FizzBuzz 脚本运行的同时，你碰巧有另一个快速分配 PID 的程序，否在你会看到 3 个连号的 PID——83561、83562 和 83563。

这些 PID 还可以用 `os` 模块写代码来查看：

```
import os
...
# 打印子进程PID
os.getpid()
# 打印父进程PID
os.getppid()
```

程序中的每个进程用 `os.getpid()` 都应该会打印一个不同的 PID，但是用 `os.getppid()` 打印的父进程 PID 是相同的。

从纯技术角度来说，对于这个示例，有两行代码是不需要的。如果不写最后的 `join` 语句：

```
for p in processes:
    p.join()
```

父进程仍然会停止，并且会自动终止子进程。但是，如果你想在这些子进程结束之后再运行其他代码，就需要使用 `join` 语句。

例如：

```
for p in processes:
    p.start()
print('Program complete')
```

如果不写 `join` 语句，就会出现下面的结果：

```
Program complete
1
2
```

如果写了 `join` 语句，程序就会等每个子进程都完成，再运行后面的代码：

```
for p in processes:
    p.start()

for p in processes:
    p.join()
print('Program complete')

...
Fizz
99
```

```
Buzz
100
Program complete
```

如果你想彻底停止程序的运行，那么当然可以用 Ctrl-C 来终止父进程。由于终止父进程就会把所有子进程一并终止，因此用 Ctrl-C 是安全的，不用担心会意外遗漏某个进程在后台运行。

16.3.1 多进程抓取

可以修改多线程维基百科抓取示例，用独立的进程来替代独立的线程：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import random

from multiprocessing import Process
import os
import time

visited = []
def get_links(bs):
    print('Getting links in {}'.format(os.getpid()))
    links = bs.find('div', {'id':'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))
    return [link for link in links if link not in visited]

def scrape_article(path):
    visited.append(path)
    html = urlopen('http://en.wikipedia.org{}'.format(path))
    time.sleep(5)
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    print('Scraping {} in process {}'.format(title, os.getpid()))
    links = get_links(bs)
    if len(links) > 0:
        newArticle = links[random.randint(0, len(links)-1)].attrs['href']
        print(newArticle)
        scrape_article(newArticle)

processes = []
processes.append(Process(target=scrape_article, args=('/wiki/Kevin_Bacon',)))
processes.append(Process(target=scrape_article, args=('/wiki/Monty_Python',)))

for p in processes:
    p.start()
```

这里我们同样用 `time.sleep(5)` 人工降低爬虫抓取的速度，因为这只是作为一个示例，所以没必要给维基百科服务器造成太大负担。

示例将用户定义的 `thread_name`（作为参数传递）替换成了 `os.getpid()`，后者不仅不需要作为参数传递，而且可以在任何时候获取。

输出结果如下：

```
Scraping Kevin Bacon in process 84275
Getting links in 84275
/wiki/Philadelphia
Scraping Monty Python in process 84276
Getting links in 84276
/wiki/BBC
Scraping BBC in process 84276
Getting links in 84276
/wiki/Television_Centre,_Newcastle_upon_Tyne
Scraping Philadelphia in process 84275
```

理论上，用独立的进程抓取比用独立的线程抓取要快，主要有两个理由。

- 进程不受 GIL 的限制，可以同时运行同一行代码，同时调整同一个对象（其实是同一个对象的多个实例化）。
- 进程可以在多个 CPU 核心上运行，如果每个进程或线程需要消耗大量的处理器资源，这可能会提升运行速度。

不过，这些优点也伴随着一大缺点。在之前的示例程序中，所有已发现的 URL 都被存储在全局的 visited 列表中。当你用多线程的时候，这个列表是由所有线程共享的；一个线程在没有遇到少见的竞争条件时，不能访问其他线程已经访问过的网页。但是，每个进程现在拥有各自独立的已访问列表，可以自由访问其他进程已经访问过的网页。

16.3.2　进程间通信

由于每一个进程各自使用独立的内存，因此如果它们之间需要通信就会有麻烦。

调整前面的示例，打印当前已经访问的列表结果，就可以看到这个问题：

```
def scrape_article(path):
    visited.append(path)
    print("Process {} list is now: {}".format(os.getpid(), visited))
```

输出如下所示：

```
Process 84552 list is now: ['/wiki/Kevin_Bacon']
Process 84553 list is now: ['/wiki/Monty_Python']
Scraping Kevin Bacon in process 84552
Getting links in 84552
/wiki/Desert_Storm
Process 84552 list is now: ['/wiki/Kevin_Bacon', '/wiki/Desert_Storm']
Scraping Monty Python in process 84553
Getting links in 84553
/wiki/David_Jason
Process 84553 list is now: ['/wiki/Monty_Python', '/wiki/David_Jason']
```

但是，有一种方法可以让同一台机器上的进程互相通信，那就是用 Python 的两个对象：队列和管线（pipe）。

这里的**队列**和之前的线程队列类似。信息由一个进程加入，再由另一个进程移除。当信息被移除之后，就会从队列中消失。由于队列被设计成一种“临时数据传输”的方法，因此它们不适合存储像“已经访问的网页列表”这样的静态引用（static reference）。

如果将网页的静态列表替换成某种抓取委托器（delegator）会怎样呢？爬虫以待抓取网页路径的形式（例如 /wiki/Monty_Python），从一个队列中获取一个任务，抓取结束后再将一个“已发现 URL”的列表返回到另一个独立的队列中，这个队列将由抓取委托器来处理，这样就只有新的 URL 会被添加到第一个任务队列中。

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import random
from multiprocessing import Process, Queue
import os
import time


def task_delegator(taskQueue, urlsQueue):
    # 为每个进程初始化一个任务
    visited = ['/wiki/Kevin_Bacon', '/wiki/Monty_Python']
    taskQueue.put('/wiki/Kevin_Bacon')
    taskQueue.put('/wiki/Monty_Python')

    while 1:
        # 检查urlsQueue中是否存在新链接需要处理
        if not urlsQueue.empty():
            links = [link for link in urlsQueue.get() if link not in visited]
            for link in links:
                # 向taskQueue中增加新链接
                taskQueue.put(link)

def get_links(bs):
    links = bs.find('div', {'id':'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))
    return [link.attrs['href'] for link in links]

def scrape_article(taskQueue, urlsQueue):
    while 1:
        while taskQueue.empty():
            # 如果任务队列为空，休息100毫秒
            # 这种情况应该极少发生
            time.sleep(.1)
        path = taskQueue.get()
        html = urlopen('http://en.wikipedia.org{}'.format(path))
        time.sleep(5)
        bs = BeautifulSoup(html, 'html.parser')
        title = bs.find('h1').get_text()
```

```
        print('Scraping {} in process {}'.format(title, os.getpid()))
        links = get_links(bs)
        # 发送这些链接到委托器进行处理
        urlsQueue.put(links)


processes = []
taskQueue = Queue()
urlsQueue = Queue()
processes.append(Process(target=task_delegator, args=(taskQueue, urlsQueue,)))
processes.append(Process(target=scrape_article, args=(taskQueue, urlsQueue,)))
processes.append(Process(target=scrape_article, args=(taskQueue, urlsQueue,)))

for p in processes:
    p.start()
```

这个爬虫和原来的爬虫在结构上有差异。每个进程或线程不再是从它们各自被分配的起始点随机游走了，而是同心协力对网站进行完整的地毯式抓取。每个进程可以从队列中拉取任何“任务”，而不再只是自己发现的链接。

16.4 多进程抓取的另一种方法

所有多线程和多进程抓取方法都假设你需要某种“父母般的规范”来引导子线程和子进程。你可以一次性启动它们，也可以同时结束它们，还可以在它们之间传递信息或者让它们共享内存。

但是如果你的爬虫不需要这种规范，或者彼此之间不需要通信呢？那就更没有理由去使用令人抓狂的 `import _thread` 了。

举个例子，假设你想要同时抓取两个类似的网站。你已经写了一个爬虫，它可以抓取任意一个网站，只需稍微调整一下配置或者一个命令行参数。你完全可以这样做：

```
$ python my_crawler.py website1
```

```
$ python my_crawler.py website2
```

瞧瞧，这样你就实现了一个多进程网络爬虫，而且还为你的 CPU 节省了一个父进程！

当然，这种方法也有不足。如果你想在同一个网站上运行两个爬虫，那么就需要某种方法来保证它们不会抓取同一个页面。解决办法可能是创建一个 URL 规则（“爬虫 1 抓取博客页面，而爬虫 2 抓取产品页面”）或者以某种方式分割网站。

另外，你也可以通过某种中间数据库来协调两个爬虫。在抓取新链接之前，爬虫可以向数据库发送请求，询问：“这个页面抓取过吗？”爬虫用数据库作为一个进程间的通信系统。当然，如果考虑不周全，这种方法也可能会导致出现竞争条件，或者由于数据库连接速度太慢而导致反馈滞后（如果连接的是远程数据库，可能会出现此类问题）。

你可能还会发现这种方法不适合扩展。`Process` 模块可以让你动态地增加或减少抓取网站的进程数量，甚至是存储数据的进程数量。手动开启它们需要一个人手动运行脚本，或者用一个单独的管理脚本（无论是 bash 脚本、cron 计划任务，还是其他方式）来做这件事。

但是，我曾经用这种方法取得了巨大的成功。对于小型、一次性的项目，这是一种迅速获取大量信息的好方法，尤其是抓取多个网站的时候。

第 17 章

远程抓取

在上一章中，你学习了用多线程和多进程运行网络爬虫，线程间和进程间的通信在一定程度上是受限的，或者说需要仔细规划。本章将针对这个问题给出一个逻辑结论——不只是在单独的进程中运行爬虫，而是完全在单独的机器上运行爬虫。

本章内容放在后面来介绍还是比较合适的。到现在为止，你已经在自己的电脑上通过命令行运行了所有的 Python 程序。当然，你可能也安装了 MySQL，以便尝试复制真实的服务器环境。但是这和实际的服务器还是不一样的。正如一句俗话所说："如果你喜欢某个东西，就放开手。"

这一章将介绍几种方法，让程序在不同的机器上运行，或者在你的电脑上用不同的 IP 地址运行。你可能打算放弃这一章，因为你现在还不需要这些内容，但是你可能会感到惊讶，原来自己已经拥有非常容易上手的工具了（比如一些付费的 VPS 或云计算资源），而且当你停止在自己的笔记本电脑上运行 Python 爬虫后，生活会变得更加轻松。

17.1　为什么要用远程服务器

虽然使用远程服务器可能像是启动一个供广大用户使用的 Web 应用时所采取的必然步骤，但我们为个人目的创建的工具通常都必须在本地运行。启用远程平台的人通常基于两个目的：需要更强的计算能力和更大的灵活性，以及需要使用可变 IP 地址。

17.1.1　避免IP地址被封杀

创建网络爬虫的第一原则是：几乎一切都可以伪造。你可以用非本人的邮箱发送邮件，通

过命令行自动化鼠标的行为，或者通过 IE 5.0 浏览器耗费网站流量来吓唬网管。

但是有一样东西是不能作假的，那就是你的 IP 地址。任何人都可以用这个地址给你写信：美国华盛顿特区宾夕法尼亚大道西北 1600 号，总统，邮编 20500。但是，如果这封信是从新墨西哥州的阿尔伯克基市寄来的，那么你可以肯定给你写信的不是美国总统。[1]

为阻止网站被抓取而做出的努力主要集中在识别人类与机器人的行为差异上面。封杀 IP 地址这种矫枉过正的行为，就好像农民不靠喷农药给庄稼杀虫，而是直接用火烧农田彻底解决问题。它是最后一步棋，不过是一种非常有效的方法，只要忽略危险 IP 地址发来的数据包就可以了。但是，使用这种方法会有以下几个问题。

- IP 地址访问列表很难维护。虽然大型网站通常都会用自己的程序自动管理 IP 地址访问列表（机器人封杀机器人），但是至少需要有人偶尔检查一下列表，或者至少要监控问题的增长。
- 因为服务器需要根据 IP 地址访问列表去检查每个准备接收的数据包，所以检查接收数据包时会额外增加一些处理时间。多个 IP 地址乘以海量的数据包会使检查时间呈指数级增长。为了减少处理时间和降低处理复杂度，管理员通常会对 IP 地址进行分组管理并制定相应的规则，比如如果这组 IP 中有一些危险分子，就“把这个区间的所有 256 个地址全部封杀”。于是产生了下一个问题。
- 封杀 IP 地址可能会将“好人”也封杀了。例如，当我在美国麻省欧林工程学院读本科的时候，有个同学写了一个可以在 http://digg.com/ 网站（在 Reddit 流行之前大家都用 Digg）上对热门内容进行投票的软件。这个软件的服务器 IP 地址被 Digg 封杀了，导致整个学校宿舍都不能访问这个网站了。于是这个同学就把软件移到了另一个服务器上，而 Digg 自己却失去了许多主要目标用户的访问量。

虽然有这些缺点，但封杀 IP 地址依然是一种十分常用的手段，服务器管理员用它来阻止可疑的网络爬虫入侵服务器。如果一个 IP 地址被封杀了，那么唯一真正的解决方案就是从不同的 IP 地址进行抓取。你可以将你的爬虫部署到一个新的服务器，或者通过使用 Tor 这样的工具将你的数据请求路由分发到不同的服务器。

17.1.2 移植性与扩展性

有些任务要想通过个人电脑连网来完成会十分困难。即使你并不想给任何一个网站增加较大的负载，但是如果你从很多网站收集数据，也会需要更快的网速以及更多的存储空间。

另外，自己电脑上的计算资源释放之后，你就可以做更重要的事了（玩魔兽，看电影，LOL）。你也不用担心电费和网速了（在星巴克启动你的应用，合上笔记本电脑离开，一

注 1：从技术上说，IP 地址是可以通过发送数据包进行伪装的，这是一种分布式拒绝服务攻击（distributed denial of service，DDoS）技术，攻击者不需要关心接收的数据包（这样发送请求的时候就可以使用假 IP 地址）。但是网页抓取是一种需要关心服务器响应的行为，所以我们认为 IP 地址是不能造假的。

切都可以安全地运行)，你还可以在任何有网络连接的地方访问你已收集的数据。

如果你的应用需要非常大的计算能力，亚马逊 AWS 的一个超大计算实例也不能满足你的需求，那么你可以看看**分布式计算**（distributed computing）。这种方法可以让多个机器并发执行并完成你的任务。一个简单的例子是你可以用一台机器来抓取一些网站，再用另一台机器抓取另一些网站，最后再把收集的数据存储在同一个数据库里。

当然，如前几章中指出的，很多应用都在重复 Google 搜索干的事情，但是没有几个程序可以达到 Google 搜索的运行规模。分布式计算是计算机科学中一个庞大的领域，超出了本书的介绍范围。但是，学习如何在远程服务器上启动你的应用是必要的第一步，之后你一定对当今计算机的能力感到无比惊讶。

17.2　Tor代理服务器

The Onion Router（洋葱路由器，Tor）网络是一种 IP 地址匿名手段。Tor 是一个由志愿者服务器构成的网络，通过由不同服务器构成的多个层（就像洋葱）把客户端包在最里面。数据进入该网络之前会被加密，因此任何服务器都不能偷取通信数据。另外，虽然每一个服务器的入站和出站通信都可以被查到，但是要想查出通信的真正起点和终点，必须知道整个通信链路上所有服务器的入站和出站通信细节，而这基本上是不可能的。

Tor 是人权工作者和政治避难人员与记者通信的常用手段，得到了美国政府的大力支持。当然，它也常被用于非法活动，所以也是政府盯防的目标（虽然目前的盯防并不是很成功）。



Tor 匿名的局限性

虽然本书中用 Tor 的目的是改变 IP 地址，而不是实现完全匿名，但有必要关注一下 Tor 匿名方法的能力和不足。

虽然 Tor 网络可以让你访问网站时显示的 IP 地址是一个不能跟踪到你的 IP 地址，但是你在网站上留给服务器的任何信息都会暴露你的身份。例如，你登录 Gmail 账号后再用 Google 搜索，那些搜索历史就会和你的身份绑定在一起。

另外，登录 Tor 的行为也可能让你的匿名状态处于危险之中。2013 年 12 月，一个哈佛大学本科生想逃避期末考试，就用一个匿名邮箱账号通过 Tor 网络给学校发了一封炸弹威胁信。结果哈佛大学的 IT 部门通过日志查到，在炸弹威胁信发来的时候，Tor 网络的流量只来自一台机器，而且是一个在校学生注册的。虽然他们不能确定流量的最初源头（只知道是通过 Tor 发送的），但是“作案”时间和注册信息证据充分，而且那个时间段内只有一台机器是登录状态，这就有充分的理由起诉那名学生了。

登录 Tor 网络不是一种自动的匿名措施，也不能让你在互联网上为所欲为。虽然它是一个实用的工具，但是使用的时候一定要谨慎、清醒，并且遵守道德规范。

要想在 Python 里使用 Tor，需要先安装并运行 Tor，下一节将介绍。Tor 服务很容易安装和开启。只要去 Tor 下载页面下载并安装，打开后连接就可以。不过要注意，当你用 Tor 的时候网速会变慢。这是因为代理有可能要先在全世界的网络上转几次才能到达目的地！

PySocks

PySocks 是一个非常简单的 Python 代理服务器通信模块，它可以和 Tor 配合使用。你可以从它的网站下载它，或者使用任何第三方模块管理器安装。

这个模块的用法很简单，示例代码如下所示。运行这段代码的时候，Tor 服务必须运行在 9150 端口（默认值）上：

```
import socks
import socket
from urllib.request import urlopen

socks.set_default_proxy(socks.SOCKS5, "localhost", 9150)
socket.socket = socks.socksocket
print(urlopen('http://icanhazip.com').read())
```

网站 http://icanhazip.com/ 只显示与服务器相连的客户端的 IP 地址，可以用来测试 Tor 是否正常运行。当程序执行之后，显示的 IP 地址就不是你原来的 IP 地址了。

如果你想在 Tor 里面用 Selenium 和 PhantomJS，根本不需要 PySocks，只要保证 Tor 在运行，然后增加 `service_args` 参数设置代理端口，让 Selenium 通过端口 9150 连接网站就可以了：

```
from selenium import webdriver
service_args = [ '--proxy=localhost:9150', '--proxy-type=socks5', ]
driver = webdriver.PhantomJS(executable_path='<path to PhantomJS>',
                             service_args=service_args)

driver.get('http://icanhazip.com')
print(driver.page_source)
driver.close()
```

和之前一样，这个程序打印的 IP 地址也不是你原来的 IP 地址，而是你通过 Tor 客户端获得的 IP 地址。

17.3 远程主机

一旦你使用信用卡，完全匿名的效果就消失了，但是把网络爬虫放在远程主机上可以大幅提升它们的运行速度。这是因为你不仅可以自由购买服务器的使用时间，使用更强大的机器，而且网络连接在到达目的地之前也不需要在 Tor 网络中长途跋涉。

17.3.1　从网站主机运行

如果你拥有个人网站或公司网站，那么你可能已经知道如何使用外部服务器运行你的网络爬虫了。即使是一些相对封闭的 Web 服务器，没有可用的命令行接入方式，你也可以通过 Web 界面对程序进行控制。

如果你的网站部署在 Linux 服务器上，该服务器上应该已经运行了 Python。如果你用的是 Windows 服务器，可能就没那么幸运了，你需要仔细检查一下 Python 有没有安装，或者问问网管可不可以安装。

大多数小型网络主机都会提供一个叫 cPanel 的软件，用来提供网站和后台服务的基本管理功能和信息。如果你接入了 cPanel，就可以设置 Python 在服务器上运行——进入“Apache Handlers”，然后增加一个 handler（如还没有的话）：

```
Handler: cgi-script
Extension(s): .py
```

这会告诉服务器所有的 Python 脚本都将作为一个 **CGI 脚本**运行。CGI 就是**通用网关接口**（Common Gateway Interface），是任何一个可以在服务器上运行，并且能动态地生成内容并显示在网站上的程序。把 Python 脚本显式地定义成 CGI 脚本，就是给服务器权限去执行 Python 脚本，而不只是在浏览器上显示它们或者让用户下载它们。

写完 Python 脚本后上传到服务器，然后把文件权限设置成 755，让它可执行。通过浏览器找到程序上传的位置（也可以写一个爬虫来自动做这件事情）就可以执行程序。如果你担心在公共领域执行脚本不安全，可以采取以下两种方法。

- 把脚本存储在一个隐晦或深层的 URL 里，确保其他 URL 链接都不能接入这个脚本，这样可以避免搜索引擎发现它。
- 用密码保护脚本，或者在执行脚本之前用密码或加密令牌进行确认。

确实，通过这些原本用来显示网站的服务来运行 Python 脚本有点儿复杂。比如，你可能会发现网络爬虫运行时网站的加载速度变慢了。其实，在整个抓取任务完成之前，页面都不会加载（得等到所有 `print` 语句的输出内容都显示完）。这可能需要几分钟，几小时，甚至永远也完成不了，要看程序的具体情况了。虽然它最终一定能完成任务，但是你可能想看到实时的结果，这样就需要一台真正的服务器了。

17.3.2　从云主机运行

以前，程序员会为了在计算机上运行或者存储自己的程序而付费。个人电脑发明之后，这似乎没必要了——人们可以直接在自己的电脑上写程序并运行。现在，应用程序的计算需求已经超越了微处理器的发展速度，于是程序员又开始为计算能力付费了。

但是，这一次用户不再为单台物理机器的计算能力付费，而是为多台机器总共的计算能力付费。这种云状计算系统的计算能力可以按使用时间进行付费。例如，当计算的低成本比即时性更重要时，亚马逊的 EC2 允许用户使用“竞价型实例”（spot instance），可以先竞价再使用云计算服务。

计算实例还可以进行定制，也可以根据应用程序的实际需求进行设置，选项有“高内存”“快速计算”“大容量存储”。虽然网络爬虫不需要很多内存，但是你可能需要较大的存储空间或快速的计算能力来实现爬虫的更多功能。如果你要做大量的自然语言处理、OCR 或者路径查找（就像“维基百科六度分隔理论”问题）之类的工作，选择“快速计算”实例就可以。如果你要抓取大量数据，存储许多文件，或者进行大数据分析，可能就需要用带大容量存储的计算实例了。

虽然云计算的花费可能是个无底洞，但是写作本书的时候，启动一个计算实例最便宜只要每小时 1.3 美分（亚马逊 EC2 的 micro 实例，其他实例会更贵），Google 最便宜的计算实例是每小时 4.5 美分，最少需要用 10 分钟。考虑计算能力的规模效应，从大公司购买一个小型云计算实例的费用跟自己买一台专业实体机的费用差不多——不过用云计算不需要雇人去维护设备。

显然，一步一步设置和运行云计算实例的教程超出了本书介绍范围，不过你自己其实不需要这类教程。亚马逊和 Google（还有不计其数的小公司）的云计算产品正在激烈地竞争，它们已经尽量简化新实例的设置步骤，你只需填个应用名称，提供一下信用卡号就可以了。写作本书的时候，亚马逊和 Google 还为新用户提供了价值几百美元的免费计算时间。

设置好计算实例之后，你就有了新 IP 地址、用户名，以及可以通过 SSH 连接实例的公私密钥了。后面要做的事情和你在实体服务器上做的一样——当然，你不再需要担心硬件维护，也不用运行复杂的监控工具了。

对于紧急且复杂的任务来说，特别是如果你缺乏处理 SSH 和密钥对的经验，我发现 Google 的云平台（Google’s Cloud Platform）实例更容易立刻建立并运行起来。它的启动器很简单，并且在启动后还有一个按钮可以用来在浏览器中查看 SSH 终端，如图 17-1 所示。

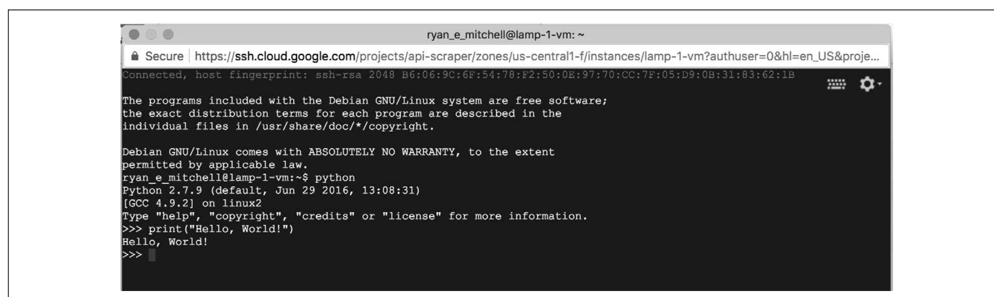


图 17-1：正在运行的 Google 云平台 VM 实例的基于浏览器的终端

17.4 其他资源

很多年以前，“在云端”运行基本上是那些既懂理论又具有服务器运维经验的人之间的高谈阔论。但是今天，由于云计算技术的不断普及，以及云计算供应商之间的竞争，云计算工具已经有了极大的改善。

如果你想创建规模更大或更复杂的爬虫，在创建云计算平台以收集和存储数据时，可能还需要一些参考资料。

Marc Cohen、Kathryn Hurley 和 Paul Newson 合著的 *Google Compute Engine* 是通过 Python 和 JavaScript 使用 Google 云计算平台的第一手资料。书中不仅介绍了 Google 的用户界面，还介绍了命令行和脚本工具，你可以利用它们增强你的应用的灵活性。

如果你更喜欢使用亚马逊的产品，Mitch Garnaat 的 *Python and AWS Cookbook* 是一本非常实用的手册，可以让你顺利启动 AWS 服务，还会告诉你如何创建并运行一个可扩展的应用。

第 18 章

网页抓取的法律与道德约束

2010 年，软件工程师 Pete Warden 构建了一个网络爬虫来从 Facebook 上收集数据。他一共收集了大约两亿名 Facebook 用户的用户名、位置、好友和兴趣爱好等信息。当然，Facebook 发现了这一行为，并给他发了一封勒令停止通知函，他照做了。有人问他为什么要依从 Facebook 的要求，他说："大数据虽然很便宜，但律师费可不便宜。"

在这一章里，我们将介绍美国与网页抓取相关的法律（以及一些国际法），并学习如何分析网页抓取行为的法律和道德约束。

在阅读下面的内容之前，希望你能理解：我是一名软件工程师，不是律师。不要把在本章或本书其他章节学到的法律知识看成专业的法律意见或规范。虽然我认为自己有足够的能力，可以讨论网页抓取行为的法律和道德约束，但是在做那些可能要承担法律责任的网页抓取项目之前，你还是应该咨询一下律师，而不是软件工程师。

本章的目的是为你提供一个框架，便于你理解和讨论网页抓取的各种合法性问题，例如知识产权、未授权的计算机访问和服务器的使用，但是本章内容并不能作为实际的法律建议。

18.1　商标、版权、专利

现在，我们开始上知识产权第一课！知识产权有 3 种基本类型：商标（用 TM 或 ® 表示）、版权（用 © 表示）和专利（有时会用文字说明某发明受专利保护或注明专利号，但通常没有任何说明）。

专利只是用来声明发明的所有权。图片、文字和任何信息本身不能获得专利权。虽然有些专利（比如软件专利）并不像我们通常理解的“发明创造”那样是有形的，但是要注意，获得专利权的是这些无形的东西（技术），而不是专利报告中的内容。除非你利用抓取来的设计图构建什么，或者有人为某种网页抓取方法获得了专利保护，否则你不太可能在网页抓取时侵犯他人的专利权。

虽然商标也不太可能成为问题，但还是需要注意的。美国专利商标局对商标的定义如下：

> **商标**（trademark）是一个单词、词组、符号和 / 或设计，用来标识和区分一种商品的来源。**服务标识**（service mark）是一个单词、词组、符号和 / 或设计，用来标识和区分一种服务而非商品的来源。术语“商标”通常既可表示商标，也可表示服务标识。

除了当我们提到商标时通常会想到的传统的单词 / 符号商标，其他的描述性特征也可以作为商标。比如，容器的外形（可口可乐的瓶子），或者一种颜色（美国欧文斯科宁的 Pink Panther 玻璃纤维隔热层的粉色）。

和专利不同，商标的所有权很大程度上由使用场景决定。比如，如果我想在博客里发一篇带可口可乐图标的文章，我完全可以这样做（只要我没有暗示我的博文是可口可乐赞助或发布的就行）。但是，如果我想制造一种新的软饮料，在外包装上使用可口可乐的图标，那明显就是侵犯了可口可乐的商标权。同样道理，虽然我可以把饮料外包装涂成 Pink Panther 的粉色，但是我不能用同样的颜色发行一款新的家用隔热层产品。

版权法

商标和专利有一个共同点，就是它们必须正式注册才能得到认可。与一般认识不同的是，受版权保护的材料并不需要注册。究竟是什么使得图像、文字、音乐等拥有版权呢？并不是说在网页下面加上“保留所有权利”（All Rights Reserved）就拥有了版权，也不是说“出版发行的”就拥有版权，而“未出版发行的”就没有。任何材料，只要你创作出来，它就会自动受到版权法的保护。

《保护文学和艺术作品伯尔尼公约》是 1886 年由瑞士政府在伯尔尼首次公布的版权国际标准。这个公约的基本含义是所有成员国都必须像对待自己国家公民的作品一样，对其他成员国公民的作品进行版权保护。其实，这就是说作为一个美国公民，如果你涉嫌抄袭一个法国公民的作品，也要承担法律责任（反之亦然）。

显然，版权是网络爬虫需要关注的内容。如果我抓取别人的博客内容然后放到自己的博客上，我就可能会惹上官司。不过，我有几层保护，可以根据博客抓取项目的实际影响，帮我进行辩护。

首先，版权保护只涉及有创造性的作品，而不涉及统计数据或事实。好在许多网络爬虫抓取的都是事实和统计数据。虽然用一个网络爬虫从网络上收集诗歌，然后显示在你自己的网站上有可能是违反版权法的，但是如果它收集不同时间段发表的诗歌数量就不违法了。诗歌是一种创造性作品，但是按月对网站上发表的诗歌进行字数统计就没什么创造性了。

如果数据是公司发布的价格、高管的姓名或者其他事实性的信息，那么即使完全照搬（不是根据抓取的原始数据进行整合或计算）也不会违反版权法。

按照《数字千年版权法》（Digital Millennium Copyright Act，DMCA），即使是有版权的内容也可以以合理理由直接使用。DMCA 列举了一些自动处理版权内容的规则。DMCA 非常长，包含了从电子书到电话的许多细则。但是，有两点与网页抓取相关。

- 根据“安全港”保护原则，如果你从一个你有理由相信只包含无版权材料的数据源抓取数据，但是有人曾向该数据源提交过有版权的材料，那么只要你在收到通知后把有版权的材料删除，就可以免责。
- 你不能为了收集信息而故意绕开安全措施，比如密码保护。

此外，DMCA 还承认《美国法典》下的“合理使用”条款适用，根据“安全港”保护原则，如果受版权保护的材料被合理地使用，DMCA 可能不会发出删除（take-down）通知。

总之，未经作者或版权所有者授权，你不可以直接发表有版权的材料。如果你以数据分析为目的，把可以自由访问的有版权的材料保存在自己的非公开数据库中，这是合法行为。如果你把数据展示到网站上供人们浏览或下载，就不算合法了。如果你分析数据库里的数据，然后发布作品的字数统计信息、按作品数量对作者排序，或发布其他的数据分析结果，这是合法行为。如果你还引用了一些原文或简单的样本数据来阐述自己的观点，也是可以的，但是使用之前最好看看《美国法典》里的“合理使用”条款。

18.2　侵害动产

侵害动产与我们常识中的“违法”有着本质的区别，动产的范围不包括不动产和土地，而是指那些可移动的财产（比如服务器）。如果接入那些不允许你接入或使用的财产，就会侵害动产。

在云计算时代，人们可能不把 Web 服务器看作一种真实有形的资源。但其实服务器不仅由许多昂贵的组件构成，而且它们还需要空间存放、监控、制冷，以及大量的电力供应。据估计，全球 10% 的电力都是由计算机消耗的。[1]（如果你自己的电费构成并非如此，可以考虑一下 Google 庞大的服务器农场，每一座农场都需要与大型电站连接。）

注 1：Bryan Walsh,“The Surprisingly Large Energy Footprint of the Digital Economy [UPDATE]”, TIME.com, August 14, 2013.

虽然服务器是很昂贵的资源，但是从法律的角度看，一个非常有趣的现象是，网站管理员非常希望人们使用他们的资源（即接入他们的网站），但同时又不希望资源被过快地消耗掉。通过浏览器看一下网站可以，但是发动大规模的 DDoS 攻击显然就不允许了。

只有满足下列 3 个条件，网络爬虫的行为才构成侵害动产。

缺少许可

由于 Web 服务器对所有人开放，所以它们一般也会向网络爬虫"提供许可"。但是，很多网站的服务协议条款都明确地禁止使用爬虫。另外，任何勒令停止通知函显然撤销了这类许可。

造成实际的损害

服务器是很昂贵的。除了服务器成本，如果你的爬虫把网站拖垮了，或者限制了网站为其他用户提供服务的能力，这些都算是你对网站造成的"损害"。

故意而为

这个，你懂的！

只有 3 个条件都满足才算是侵害动产。然而，如果你违反了服务协议，但并未造成实际的损害，不要以为你就不算违法。可能你的行为已经违法了版权法、DMCA、《计算机欺诈与滥用法》（The Computer Fraud and Abuse Act，CFAA，后面会详细介绍），或者其他可以处理网络爬虫犯罪行为的法律。

请限制你的爬虫

过去，Web 服务器比个人电脑要强大得多。其实，"服务器"的部分定义就是指"大型计算机"。而现在情况似乎反过来了。比如，我的个人电脑拥有一个 3.5GHz 处理器和 8G 内存。亚马逊的一个中等云计算实例（写作本书的时候）却只有 3GHz 处理器和 4G 内存。

如果网速正常，还有一台可以持续抓取的专用设备，即使是一台个人电脑也可以给许多网站造成沉重负担，甚至可以对网站造成严重损害或者直接把网站拖垮。除非出现了紧急医疗事故，而唯一的援救方法是在两秒内收集《阿周真人秀》（Joe Schmo）网站上所有的搞笑视频，否则真的没有理由去损害别人的网站。

一直被盯着看的机器人是永远不会完成任务的（抓取总是需要很长时间）。有时候最好让爬虫在午夜运行，而不是在下午或者傍晚运行，原因如下。

- 如果你有大约 8 个小时的时间，即使抓取一页需要 2 秒，你也可以抓取 14 000 多个页面。当时间不怎么紧张的时候，没必要加快爬虫的抓取速度。

- 假如网站的目标访客和你在同一时区（如果不在同一时区，可以相应地调整时间），那么夜间网站流量可能会少很多，这就意味着你的抓取行为不会影响网站高峰期的运行了。
- 你可以在爬虫抓取网站的时候睡觉，不必为了看到新信息而不断地翻日志。想想看，第二天早上睡醒的时候崭新的数据就摆在面前，得有多么惬意啊！

再想象一下下面 3 种场景：

- 你有一个网络爬虫遍历了《阿周真人秀》网站，收集了一些或全部的数据；
- 你有一个网络爬虫遍历了几百个小网站，收集了一些或全部的数据；
- 你有一个网络爬虫遍历了一个超大型网站，比如维基百科。

在第一个场景中，最好让爬虫在深夜慢慢地运行。

在第二个场景中，最好以循环的方式快速地抓取每个网站，而不是一次一个慢慢地抓取。根据你要抓取的网站数量进行合理安排，你就可以以最快的快速（取决于网络连接和机器）收集数据，而且对每个远程服务器造成的负载也比较合理。为实现这种循环抓取方式，你可以采用多线程（每个线程抓取一个网站，可以暂停），也可以用 Python 列表来跟踪网站。

在第三个场景中，可能你的网络连接和个人电脑对维基百科这样的超大型网站造成的负载不会引起对方的注意。但是，如果你用分布式网络设备抓取，显然就不是一回事儿了。请谨慎使用分布式网络设备，最好问问对方允不允许这么做。

18.3 计算机欺诈与滥用法

在 20 世纪 80 年代早期，计算机开始从学术领域走向商业世界。病毒和蠕虫不再仅仅被认为是麻烦事（或者一种业余爱好），而是可能导致实际财务损失的严重犯罪事件。为此，美国联邦政府在 1986 年出台了《计算机欺诈与滥用法》。

尽管你可能会认为《计算机欺诈与滥用法》只是针对那些发布病毒的恶意黑客，但其实它对网络爬虫也有很大的影响。想象一下，一个爬虫在网上寻找采用简单易猜密码的登录表单，对网站进行暴力破解，或者收集不小心置于隐蔽但公开位置的政府机密。根据《计算机欺诈与滥用法》，这些行为都是非法的。

《计算机欺诈与滥用法》定义了 7 种主要犯罪行为，总结如下。

- 明知没有授权，却侵入美国政府的计算机，并获取信息。
- 明知没有授权，却侵入计算机，并获取财务信息。
- 明知没有授权，却侵入美国政府的计算机，影响政府计算机的使用。
- 为了诈骗的目的故意侵入任何受保护的计算机。

- 在未经授权的情况下，故意侵入一台计算机并导致计算机损坏。
- 分享或买卖美国政府使用的计算机或者影响州际或国际商务往来的计算机的密码或授权信息。
- 试图通过破坏或威胁破坏任何受保护的计算机，敲诈钱财或“任何有价值的东西”。

总之，远离那些受保护的计算机，不要接入没有授权的计算机（包括 Web 服务器），尤其要避开政府或财务计算机。

18.4　robots.txt和服务协议

从法理上说，网站的服务协议和 robots.txt 是很有趣的。如果一个网站允许公众接入，那么网站管理员对哪些软件可以接入而哪些软件不可以接入的限制是不合理的。如果网站管理员对你说，“你用浏览器访问网站没问题，但是你用自己写的程序访问它就不行”，这就不太靠谱了。

大多数网站在每页的页脚都有自己的服务协议链接。服务协议不仅包含网络爬虫和自动接入的规则，而且还包括网站收集的信息类型和信息用途，通常还有一条免责声明，表明对网站提供的服务不做任何明示或默示保证。

如果你对搜索引擎优化（search engine optimization，SEO）或搜索引擎技术感兴趣，那么你可能听说过 robots.txt 文件。如果你想在任何大型网站上查找 robots.txt 文件，可以在网站根目录 http://website.com/robots.txt 找到。

robots.txt 文件的语法是在 1994 年出现的，那时搜索引擎技术刚刚兴起。当时，从整个互联网寻找资源的搜索引擎，比如 AltaVista 和 DogPile，开始和那些按照主题对网站进行分类的门户网站激烈竞争，比如 Yahoo!。互联网搜索规模的增长不仅意味着网络爬虫数量的增长，而且也意味着网络爬虫收集的信息对普通人而言的可供性大大增强了。

虽然我们今天认为这种可用性是稀松平常的，但在当时，当网站文件结构深处隐藏的信息出现在主要搜索引擎的搜索结果首页中时，有些网站管理员感到非常震惊。于是，robots.txt 文件的语法，也称为机器人排除标准（Robots Exclusion Standard），应运而生。

与通常用人类语言宽泛地讨论网络爬虫的服务协议不同，robots.txt 文件可以被自动化程序轻易地解析和使用。虽然它似乎可以一劳永逸地解决爬虫问题，但是请注意下面两点。

- robots.txt 文件的语法没有标准格式。它是一种常用并被良好遵循的规范，但是并未阻止任何人创建自己的 robots.txt 文件（且不说除非它变成主流标准，否则网络机器人就不会承认或遵循它）。尽管如此，它仍是一种被企业广泛认可的规范，主要是因为它非常简单，而且企业也没什么动力去开发自己的版本或者尝试去改进它。

- robots.txt 文件并不是一个强制性约束。它只是说“请不要抓网站的这些内容”。很多网络爬虫库都支持 robots.txt 文件（虽然这通常是个很容易修改的默认设置）。另外，按照 robots.txt 文件抓取信息比直接抓取要麻烦得多（毕竟，你需要抓取、分析并在代码逻辑中处理页面内容）。

机器人排除标准的语法很简单。和 Python 等语言一样，注释都是用 `#` 号开头，用换行符结尾，可以用在文件的任意位置。

文件的第一行非注释内容是 `User-agent:`，注明具体哪些机器人需要遵守规则。后面是一组规则，要么是 `Allow:` 要么是 `Disallow:`，决定了是否允许机器人访问网站的该部分内容。星号（`*`）是通配符，可以用于 `User-agent:`，也可以用于 URL 链接中。

如果一条规则后面跟着一个与之矛盾的规则，则按后一条规则执行。例如：

```
#Welcome to my robots.txt file!
User-agent: *
Disallow: *

User-agent: Googlebot
Allow: *
Disallow: /private
```

在这个例子中，所有的机器人都被禁止访问网站的任何内容，除了 Google 的网络机器人，它可以访问网站上除 `/private` 位置之外的所有内容。

Twitter 的 robots.txt 文件对 Google、Yahoo!、Yandex（俄罗斯著名搜索引擎）、微软，以及其他机器人或搜索引擎的访问范围都有明确的说明。Google 搜索（和其他机器人的访问范围一样）的内容如下所示：

```
#Google Search Engine Robot
User-agent: Googlebot
Allow: /?_escaped_fragment_

Allow: /?lang=
Allow: /hashtag/*?src=
Allow: /search?q=%23
Disallow: /search/realtime
Disallow: /search/users
Disallow: /search/*/grid

Disallow: /*?
Disallow: /*/followers
Disallow: /*/following
```

注意，Twitter 限制访问其网站中有 API 的部分。因为 Twitter 有一个管理良好的 API（并且可以通过授权赚到钱），所以禁止任何“自制 API”通过独立抓取其网站来收集信息对 Twitter 最为有利。

虽然看到一个指明爬虫抓取范围的文件让人感觉很憋屈，但是它其实可以成为网络爬虫开发的指示灯。如果你发现一个 robots.txt 文件禁止抓取网站上某个部分的内容，那么基本可以确定网管同意你抓取其他部分的所有内容（如果他们不愿意让你抓取，就会在 robots.txt 文件中明令禁止了）。

例如，维基百科的 robots.txt 文件中适用于一般网络爬虫（并非搜索引擎）的部分非常宽容。它甚至用人类可以阅读的文字来欢迎机器人抓取（适合我们的爬虫！），并且只禁止访问一小部分页面，比如登录页面、搜索页面和“随机词条”页面。

```
#
# Friendly, low-speed bots are welcome viewing article pages, but not
# dynamically generated pages please.
#
# Inktomi's "Slurp" can read a minimum delay between hits; if your bot supports
# such a thing using the 'Crawl-delay' or another instruction, please let us
# know.
#
# There is a special exception for API mobileview to allow dynamic mobile web &
# app views to load section content.
# These views aren't HTTP-cached but use parser cache aggressively and don't
# expose special: pages etc.
#
User-agent: *
Allow: /w/api.php?action=mobileview&
Disallow: /w/
Disallow: /trap/
Disallow: /wiki/Especial:Search
Disallow: /wiki/Especial%3ASearch
Disallow: /wiki/Special:Collection
Disallow: /wiki/Spezial:Sammlung
Disallow: /wiki/Special:Random
Disallow: /wiki/Special%3ARandom
Disallow: /wiki/Special:Search
Disallow: /wiki/Special%3ASearch
Disallow: /wiki/Spesial:Search
Disallow: /wiki/Spesial%3ASearch
Disallow: /wiki/Spezial:Search
Disallow: /wiki/Spezial%3ASearch
Disallow: /wiki/Specjalna:Search
Disallow: /wiki/Specjalna%3ASearch
Disallow: /wiki/Speciaal:Search
Disallow: /wiki/Speciaal%3ASearch
Disallow: /wiki/Speciaal:Random
Disallow: /wiki/Speciaal%3ARandom
Disallow: /wiki/Speciel:Search
Disallow: /wiki/Speciel%3ASearch
Disallow: /wiki/Speciale:Search
Disallow: /wiki/Speciale%3ASearch
Disallow: /wiki/Istimewa:Search
Disallow: /wiki/Istimewa%3ASearch
Disallow: /wiki/Toiminnot:Search
Disallow: /wiki/Toiminnot%3ASearch
```

是否遵照 robots.txt 文件的要求写网络爬虫由你自己决定，但是我强烈建议你遵守，尤其是你的爬虫不加选择地抓取网页的时候。

18.5　3个网络爬虫

因为网页抓取是一个无界限的领域，所以你很容易陷入官司之中。这一节将介绍 3 个案例，其中均涉及某种适用于网络爬虫的法律。

18.5.1　eBay起诉Bidder's Edge侵害其动产

1997 年，豆宝宝（Beanie Baby）市场依旧如火如荼，科技领域的泡沫不断膨胀，在线房屋拍卖已成为互联网上的新热点。有一家叫 Bidder's Edge 的公司创造了一种新的拍卖网站。客户不需要到各个拍卖网站上查看并对比商品价格，这个公司可以汇总所有网站上关于同一商品（比如一个流行的 Furby 娃娃或电影《辣妹世界》的光盘）的信息，然后客户就可以很方便地点击最低价的网站去购买了。

Bidder's Edge 通过很多网络爬虫实现了这一点。为了获得商品价格和信息，它们不断地向各个拍卖网站的 Web 服务器发起请求。在当时的拍卖网站中，最大的是 eBay，Bidder's Edge 每天要向 eBay 服务器请求大约 100 000 次。就算按照今天的标准，这也是很大的流量。eBay 公布的数据显示，这相当于其网站一天总流量的 1.53%，该公司自然对此感到不满。

eBay 给 Bidder's Edge 发了一封勒令停止通知函，以及一张 eBay 数据授权申请表。但是，授权谈判没成功，Bidder's Edge 仍然一意孤行，继续抓取 eBay 的数据。

虽然 eBay 封杀了 Bidder's Edge 的 169 个 IP 地址，但是 Bidder's Edge 可以通过代理服务器继续抓取（发送请求的时候显示代理服务器的 IP）。"暗战"就这样开始了。在旧 IP 被封杀之后，Bidder's Edge 不断启用新的代理服务器并购买新的 IP 地址，eBay 则被迫不断更新防火墙列表（并对每个可疑 IP 地址发送的数据包进行检查）。

最终，在 1999 年 12 月，eBay 起诉 Bidder's Edge 侵害其动产。

因为 eBay 的服务器是其拥有的真实有形的资源，它不想让 Bidder's Edge 滥用自己的资源，所以起诉对方侵害动产好像非常合理。实际上，在当代，侵害动产在网络爬虫法律案件中十分普遍，也经常被视为 IT 法律。

法院认为，eBay 需出示两方面证据才可以证明自己的动产被侵害了：

- Bidder's Edge 未经许可便使用了 eBay 资源
- eBay 确实因为 Bidder's Edge 的行为遭受了经济损失

由于之前 eBay 发过勒令停止通知函，而且 IT 日志可以显示服务器的使用情况以及相关成

本，所以 eBay 很容易就提供了证据。当然，大型法律案件都不会轻松结束：对方提起了反诉，双方聘请了多位律师，最终在 2001 年 3 月于庭外和解，赔偿金额不详。

那么，这是不是说，以后只要任何人未经授权使用他人的服务器，就是侵害动产了呢？也不一定。Bidder’s Edge 是一个极端案例：它使用了 eBay 太多的资源，导致 eBay 不得不购买更多的服务器，花更多电费，可能还要雇用更多的人进行维护（虽然 1.53% 看似并不多，但对这样的大公司来说所有加总肯定是一笔大数目）。

2003 年，加州最高法院宣判了另一个案子，Intel 公司起诉 Hamidi 失败。Intel 前雇员 Hamidi 通过 Intel 服务器向 Intel 的员工发送让 Intel 公司不爽的邮件。法院结案时说：

> Intel 败诉并不是因为通过网络发送邮件不必承担任何法律责任，而且因为在加州，如果原告不能证明自己的财产或法律权益受到了损害，那么侵害动产的民事侵权行为（不同于起诉理由）就不成立。

最后，Intel 无法向法院证明 Hamidi 向公司其他员工发送的 6 封邮件给员工造成了经济损失（有趣的是，每个员工都有一个“从 Hamidi 邮件列表中删除”选项——说明他还是挺懂规矩的）。这件事并没有给 Intel 造成任何财产损失。

18.5.2　美国政府起诉Auernheimer与《计算机欺诈与滥用法》

如果网上的信息可以让人用浏览器轻而易举地获得，那么你用自动化手段获取同样的信息就不太可能会引起联邦调查局调查你。但是，如果一个非常细心的人在网站上发现了一个极小的安全漏洞，再使用网络爬虫自动化抓取网站，那么这个极小的安全漏洞就会变得越来越大并且非常危险，被联邦调查局调查就很正常了。

2010 年，Andrew Auernheimer 和 Daniel Spitler 在 iPad 上发现了一个新功能。当你用 iPad 访问 AT&T 网站的时候，AT&T 会跳转到一个包含你的 iPad 唯一 ID 号的链接：

```
https://dcp2.att.com/OEPClient/openPage?ICCID=<idNumber>&IMEI=
```

这个页面包括一个登录表单，上面显示了对应 ID 号的用户的邮箱地址，用户只要输入密码就可以登录他们的账号了。

虽然有大量可能的 ID 号，但只要有足够多的爬虫，用一串随机数迭代，就可以收集邮箱地址。通过提供这个方便的登录功能，AT&T 基本上就把用户的邮箱地址公布到网络上了。

Auernheimer 和 Spitler 创建了一个爬虫，一共收集了 114 000 个邮箱地址，里面包含知名人士、企业 CEO 和政府官员的私人邮箱地址。Auernheimer 将该邮箱地址列表以及获取列表的方法发送给了高客传媒（Gawker Media），高客传媒也很给力，在自己的网站发布了头条消息“苹果最严重的安全漏洞：114 000 个 iPad 用户信息被曝”（不过没有公布邮箱列表）。

2011 年 6 月，Auernheimer 的家突然遭到 FBI 搜查，FBI 索要邮箱地址，不过最终以贩毒罪逮捕了他。2012 年 11 月，他因未经授权侵入计算机被判欺诈与共谋罪，被判入狱 41 个月，并被判处罚金 73 000 美元。

他的案子引起了民事律师 Orin Kerr 的关注，Kerr 加入了他的律师团队，将案件上诉至美国联邦第三巡回上诉法院。2014 年 4 月 11 日（这类法律程序耗时都比较长），第三巡回上诉法院接受上诉，法院的意见是：

> Auernheimer 在第一法院的定罪必须撤销，因为根据《计算机欺诈与滥用法》，18 U.S.C. § 1030(a)(2)(C)，访问公开可访问的网站并非未经授权的访问。AT&T 并没有使用密码或任何其他保护措施来限制对其用户的邮箱地址的访问。AT&T 主观上希望外人不会偶然看到敏感数据，以及 Auernheimer 将访问夸张地描述为"偷窃"，这都不重要。AT&T 的服务器配置使得信息向所有人公开，就是授权公众查看信息。根据《计算机欺诈与滥用法》，通过 AT&T 的公共网站获取邮箱地址是获得授权的行为，因此 Auernheimer 无罪。

于是，理智在法律体系中又一次获得了最终胜利。同一天，Auernheimer 被从监狱中释放，从此每个人都可以快乐地生活了。

虽然 Auernheimer 被认定为没有违反《计算机欺诈与滥用法》，但是他的家被 FBI 强行搜查了，他还花费了数千美元的律师费，还花了三年时间诉讼，还坐了牢。作为网络爬虫从业者，我们能从中吸取什么教训，避免类似情况发生在自己身上呢？

抓取任何敏感信息的时候，无论是个人隐私（本案例中是邮箱地址）、商业秘密还是政府机密，在向律师咨询之前，都不应该行动。即使信息是公开的，你也要想想："如果普通用户想看这些信息，可以轻松获取到吗？""这些信息是公司想让用户看的吗？"

我曾经多次给一些公司打电话，告诉他们网站和 Web 应用存在的安全隐患。这么说最合适："你好，我是一名网络专家，我在你们的网站上发现了一个潜在的安全隐患，可以把电话转接到可以处理问题的人那里吗？"对方除了立刻认可你的（白帽）黑客精神，还可能会让你免费订阅网站内容，甚至还会有现金奖励或其他好处！

另外，Auernheimer（在通知 AT&T 之前）向高客传媒发布信息，以及炫耀自己利用了安全漏洞，使得他成为了 AT&T 律师的一个特别有吸引力的目标。

如果你发现了网站的安全隐患，最好的做法就是告诉网站的所有者，而不是媒体。尤其是当网站没有及时发布补丁的时候，你可能很想写一篇博文以向世界公布。但是，你应该记住，那是网站公司该做的事情，与你无关。你最该做的就是让你的网络爬虫（还有你的业务）远离这些网站！

18.5.3　Field起诉Google：版权和robots.txt

Blake Field 是一名律师，他起诉 Google 违反了版权法，因为当他把自己的书从他的网站上删除之后，Google 还是在搜索引擎里显示了书的副本。版权法允许具有原创性作品的作者控制作品的发布渠道。Field 认为 Google 的缓存（当他把自己的书从他的网站上删除之后）侵犯了他控制作品发布渠道的权利。



Google 网络缓存

Google 网络爬虫（也叫**谷歌机器人**）抓取网站的时候，它们会为网站制作一个副本，然后放在互联网上。任何人都可以用 URL 链接接入这些缓存：

```
http://webcache.googleusercontent.com/search?q=cache:http:
//pythonscraping.com/
```

如果你搜索或抓取的网站没有了，你可以用这个方法看看是否有可用的副本。

知道 Google 的缓存功能却没有采取安全措施，这对 Field 不利。毕竟，他可以通过在网站上增加 robots.txt 文件来禁止 Google 机器人缓存他的网站，并在里面注明哪些页面可以抓取，哪些页面不能抓取。

更重要的是，法院认为，根据 DMCA 的安全港条款，Google 可以合法地缓存和显示 Field 的网站："服务提供商作为中间媒介或临时把材料存储在由其控制或操作的系统或网络上，不应当做出经济赔偿……不应当承受侵犯版权的责任。"

18.6　勇往直前

Web 一直在不断地变化。那些给我们带来了图像、视频、文字和其他数据文件的计算机技术也在不断地升级和改进。如果想紧跟技术潮流，抓取互联网数据的技术也需要随机应变。

谁知道呢？本书未来的版本可能会完全忽略 JavaScript，届时它已是一种过时的、极少使用的技术了，而重点关注 HTML8 的全息投影解析。但是，抓取网站内容的基本思路和一般方法是不会改变的。

无论现在还是将来，遇到一个网页抓取项目时，你都应该问问自己以下几个问题。

- 我需要回答或要解决的问题是什么？
- 什么数据可以帮到我？它们都在哪里？
- 网站是如何展示数据的？我能准确地识别网站代码中包含这一信息的部分吗？
- 如何定位这些数据并获取它们？
- 为了让数据更实用，应该做怎样的处理和分析？
- 怎样才能让抓取过程更好，更快，更稳定？

此外，你不仅需要掌握如何使用本书中介绍的工具，还要知道如何把它们有效地组合起来以解决更大的问题。有时，数据很容易获取，格式也很规范，用一个简单的爬虫就搞定了。有时，你可能需要仔细地思考一番才能抓取到数据。

例如，在第 11 章，我首先用 Selenium 获取在亚马逊图书预览页面中通过 Ajax 加载的图片，然后再用 Tesseract 读取图片，识别里面的文字。在“维基百科六度分隔”问题中，我先用正则表达式实现一个爬虫，把维基百科词条链接信息存储到数据库中，然后用有向图算法寻找词条“凯文·贝肯”与词条“埃里克·艾德尔”之间的最短链接路径。

在使用自动化技术抓取互联网数据时，其实很少遇到完全无法解决的问题。记住一点就行：互联网其实就是一个用户界面不太友好的超级 API。

关于作者

瑞安·米切尔（Ryan Mitchell）是美国波士顿 HedgeServ 公司的一名高级软件工程师，负责开发公司的 API 和数据分析工具。她本科毕业于美国欧林工程学院，之后在哈佛大学继续教育学院获得了软件工程硕士学位和数据科学证书。在加入 HedgeServ 公司之前，她曾在 Abine 公司构建网络爬虫和网络机器人。她还经常为零售、金融和医药行业的网页抓取项目提供咨询服务，并在美国东北大学和美国欧林工程学院担任课程顾问和兼职教员。

关于封面

本书封面上的动物是一只南非穿山甲。穿山甲是一种独居、喜欢夜间活动的哺乳动物，与犰狳、树懒、食蚁兽是近亲。它们主要分布于非洲的东部和南部。非洲还有 3 种穿山甲，均属濒临灭绝物种。

南非穿山甲一般体长 30~152 厘米，体重 1.5~33 千克。它们和犰狳类似，身上有深棕色、浅棕色或橄榄色的鳞甲。幼年穿山甲的鳞甲主要呈粉红色。受到威胁时，其尾部的鳞甲更像攻击性武器，可以砍伤攻击者。穿山甲还有一种与臭鼬类似的防御策略，可以从肛门附近的腺体中释放出一种酸性恶臭气体。这么做不仅是向潜在的攻击者发出警告，还可以标记自己的势力范围。穿山甲的肚子上并没有鳞甲，不过有一点儿毛。

和它们的近亲食蚁兽一样，穿山甲主要以蚂蚁和白蚁为食。它们异乎寻常的长舌头可以在树洞和蚂蚁窝中寻觅食物。它们的舌头比身体还长，不用的时候可以缩回胸腔里。

虽然穿山甲是独居动物，但是长大以后会居住在很深的地洞里。但是它们经常“霸占”土豚和疣猪弃用的巢穴。不过通过前肢上 3 个又长又弯的爪子，穿山甲在需要的时候为自己挖一个地洞也不成问题。

O’Reilly 图书封面上的许多动物都濒临灭绝，它们对这个世界非常重要。如果你想了解如何能够帮助它们，请参考 animals.oreilly.com。

封面图片取自 Lydekker 的 *The Royal Natural History*。



微信连接



回复“Python”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

Python网络爬虫权威指南(第2版)

作为一种采集和理解网络上海量信息的方式，网页抓取技术变得越来越重要。而编写简单的自动化程序（网络爬虫），一次就可以自动抓取上百万个网页中的信息，实现高效的数据采集和处理，满足大量数据需求应用场景。

本书采用简洁强大的Python语言，全面介绍网页抓取技术，解答诸多常见问题和误解，是掌握从数据爬取到数据清洗全流程的系统实践指南。书中内容分为两部分。第一部分深入讲解网页抓取的基础知识，重点介绍BeautifulSoup、Scrapy等Python库的应用。第二部分介绍网络爬虫编写相关的主题，以及各种数据抓取工具和应用程序，帮你深入互联网的每个角落，分析原始数据，获取数据背后的故事，轻松解决遇到的各类网页抓取问题。第2版全面更新，新增网络爬虫模型、Scrapy和并行网页抓取相关章节。

- 解析复杂的HTML页面
- 使用Scrapy框架开发爬虫
- 学习存储数据的方法
- 从文档中读取和提取数据
- 清洗格式糟糕的数据
- 自然语言处理
- 通过表单和登录窗口抓取数据
- 抓取JavaScript及利用API抓取数据
- 图像识别与文字处理
- 避免抓取陷阱和反爬虫策略
- 使用爬虫测试网站

“这本书很实用，非常适合用来解决实际问题。我就利用书中的工具和示例轻松地将一些重复性工作自动化了，进而将省下来的时间用于处理更有意思的事情。”

——Eric VanWyk
美国欧林工程学院
电子计算机工程师

瑞安·米切尔（Ryan Mitchell），数据科学家、软件工程师，有丰富的网络爬虫和数据分析实战经验，目前就职于美国格理集团，经常为网页数据采集项目提供咨询服务，并在美国东北大学和美国欧林工程学院任教。

PYTHON

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机/程序设计/Python

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版



ISBN 978-7-115-50926-0
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks